Blue Cat's Plug'n Script User Manual

"Craft your own effects and instruments"

BLUE CAT AUDIO

Copyright (c) 2014-2015 Blue Cat Audio

Table Of Content

- Introduction
  - Description
  - Features
  - System Requirements
  - Installation
  - First Launch
- Using Blue Cat's Plug'n Script
  - Introduction
  - The User Interface
  - Operation
  - Scripting Tutorial
  - Angelscript Scripting Reference
    - AngelScript Version
    - AngelScript Add- ons
    - API
    - Script Reference
  - Advanced Scripting Topics
    - Debugging Scripts
    - Optimization and Best Practices
  - Native Programming Tutorial
  - Native Programming Reference
    - API
    - C + + Script Reference
    - C Script Reference
  - Writing Your Own User Interface
  - Adding a User Manual
  - Resources
- Blue Cat Audio Plug- ins Basics
  - User Interface Basics
  - Controls
  - Keyboard
  - Mouse
  - Presets Management
  - MIDI & Automation Control
  - More
- Blue Cat's Plug'n Script Parameters
  - Input
  - Output
- Plug- in Settings
  - The Global Settings Window
  - The Preset Settings Window
- About Skins
  - Changing the Skin
  - Create a Custom Skin
- Frequently Asked Questions
- More
  - Extra Skins
  - Tutorials
  - Updates
  - Versions History

Copyright (c) 2014-2015 Blue Cat Audio

Note: *An online version of this user manual is available here.*

Copyright (c) 2014-2015 Blue Cat Audio

Introduction

Description

Blue Cat's Plug'n Script is a scripting plug- in that can be programmed to quickly build custom audio and MIDI effects or virtual instruments, without quitting your favorite DAW software.

With this plug- in, you can write your own effects with very little knowledge about programming: no need to invest time and money to learn all these plug- in formats, they are all supported here already! If you do not care about programming, Blue Cat's Plug'n Script can also be used as a regular processor with existing scripts or to get someone else write this very particular effect you have been searching for years and cannot find anywhere.

It is also possible for audio software companies to use this plug- in to quickly prototype algorithms and reduce the time required to develop audio products: writing and testing scripts is lightning fast!

This plug- in has been in our lab for several years and has been used to prototype lots of our existing plug- ins. It supports both scripts distributed as source code and pre- compiled binary scripts.

For source scripts, it uses the high performance AngelScript scripting engine, originally developed for video games, with a JIT compiler that helps compiling scripts into machine code for optimal performance. Starting with version 2, the plug- in also supports native (compiled) code and can load binary scripts built from C or C + + (or any language that can export C functions), using the same interface as the angelscript version.

The syntax of the AngelScript language is close to Java, C#, C + + or JavaScript, making it very easy to learn. It is also convenient to reuse your scripts into native code, or integrate code snippets found on the web into your script. Also, switching between angelscript and binary versions of the same script is very easy, requiring very few changes.

The plug- in includes dozens of audio and MIDI processing scripts as well as several virtual instruments and utilities to enhance your workflow. They can be used as is, or as examples to get started to write your own scripts. User scripts can be shared in the scripts repository, which also includes the factory scripts for reference.

Note: on Mac, the factory native scripts require Mac OSX 10.7 or newer.
To build complex effects chains using multiple scripts, you might be interested in Blue Cat's PatchWork, which can host multiple instances of this plug- in in series or parallel. While writing scripts, you might also want to use our analysis tools to check their effect on the signal.

***Typical applications:** Audio and MIDI FX scripting, virtual instruments programming, digital signal processing (DSP), Multi- effect processor, algorithm prototyping, learning, plug- in development.*

Features

Main Features:

- Audio and MIDI scripting plug- in.
- Build your own Audio and MIDI effects or virtual instruments.
- Simple scripting syntax, using either the AngelScript language (similar to C + +, Java or javascript) or C/ C + + for native code.
- Native binary scripts support (compiled C or C + +).
- The user interface adapts to scripts automatically. Build your own using Blue Cat's KUIML language.
- Up to 48 input parameters and 32 outputs, with full MIDI and automation control.
- Use your favorite Integrated Development Environment (IDE) for editing.
- Integrated JIT (Just In Time) compiler for optimal performance (scripts are not interpreted but compiled).
- Includes **55 scripts** with source code and **132 presets** for sound synthesis, MIDI FX, audio FX (comp, gate, tremolo echo, filters, looper...), waveform generation, file playback/ recording and more.
- Side chain input and auxiliary outputs support.

Blue Cat Audio Standards:

- Available as: Mac- AAX, Mac- AU, Mac- RTAS, Mac- VST, Win- AAX, Win- DX, Win- RTAS, Win- VST, Win x64- AAX, Win x64- DX, Win x64- VST.
- Native DSP code for optimal performance.
- Full MIDI control and automation support with silent, zipper- free parameters update, advanced response control and MIDI Learn.
- No CPU load on idle: when the plug- in is fed with silence, the processing smoothly shuts down, to optimize the CPU usage of your Digital Audio Workstation.
- Skinnable and customizable user interface with transparency management.
- Automation and MIDI output: record output parameters as automation curves or send MIDI CC messages to other plug- ins for powerful side chain effects.
- Smooth Bypass: activate/ deactivate the plug- in with absolutely no noise.
- Undo/ Redo.
- Full featured integrated presets manager.

Copyright (c) 2014-2015 Blue Cat Audio

- Surround support (up to 8 audio channels).
- Integrated undo/ redo.
- Copy/ paste the plug- in's state between instances using the system clipboard.
- Any sample rate supported.

System Requirements

Windows

- An SSE2- enabled processor (Pentium 4 or newer).
- Microsoft Windows XP, Vista, Windows 7, 8 or 10.
- Any DirectX / VST / RTAS / AAX compatible host software (32 or 64 bit).

Mac OS X

- An Intel processor.
- Mac OS X Yosemite (10.10), Mavericks (10.9), Mountain Lion (10.8), Lion (10.7), Snow Leopard (10.6), or Leopard (10.5).
- Any VST / Audio Unit (32/64- bit) / RTAS / AAX compatible application.

For more information about supported platforms, see our FAQ.

Installation

The plug- ins versions cannot be run standalone: they require a host application (see the System Requirements chapter for more information). Depending on which host application you use, you might need to install the plug- ins in different locations.

Before installing on of the plug- in versions, you will need to close all your host applications.

Windows

Install

All versions of the plug- in provide an installation program. Follow the steps of the wizard to install the software on your machine. During the installation you will be asked where you want the software to be installed. For the VST version, you should install the plug- in inside the VST plug- ins folder used by your host application(s). The default path set in the installer should work for most applications, but you should check your host software documentation to know where it looks for VST plug- ins. For other plug- in types, you should just use the standard path.

Some applications will not automatically rescan the new plug- ins, so you might have to force a refresh of the plug- ins list.

Upgrade

When a new version of the software is released, just launch the new installer: it will update the current installation.

Uninstall

To uninstall the software, simply launch the "Uninstall" program that is available in the start menu or in the configuration panel. It will take care of removing all files from your computer.

Mac

Install

On Mac the plug- ins are available as drive images. After download, double click on the file to open it. You can then drag and drop the file(s) to the shortcut that is provided within the image. It will install the software for all users on the machine.

In case you do not have admin rights on your Mac or if you want to install the software to another directory, just copy the files to the appropriate location. If required, more information is available in the README.txt file that is included in the package.

Upgrade

When a new version of the software is released, open the new image and copy the files over the previous ones. The new version will replace the older one.

Uninstall

To uninstall the software, simply remove the component(s) from the folder where you have copied them during install (move them to the trash).

If you want to completely remove all settings and configuration files, you can also remove these additional directories that may have been created on your computer:

- ~/ Library/ Preferences/ Blue Cat Audio/ [Plug- in name and TYPE], where TYPE is VST, AU, RTAS or AAX: global preferences.
- ~/ Library/ Preferences/ Blue Cat Audio/ [Plug- in name]: license information

Copyright (c) 2014-2015 Blue Cat Audio

- ~/ Documents/ Blue Cat Audio/ [Plug- in name]: user data, such as presets, additional skins and user- created plug- in data.

Just beware that these directories may contain user data that you have created. Remove these directories only if you do not want to reuse this data later.

First Launch

Blue Cat Audio plug- ins cannot be run standalone, they require a host application (see the System Requirements chapter for more information). Some host applications will require you to scan the plug- ins before they are available in the application.

If the plug- in is not available in the application, please check that it has been installed in the appropriate directory (with no host application running), and that the host application has scanned it.

Copyright (c) 2014-2015 Blue Cat Audio

Using Blue Cat's Plug'n Script

Introduction

The idea behind Blue Cat's Plug'n Script is to provide a simple way to create your own audio or MIDI processing algorithms without much knowledge about programming.

Effect or Instrument

Two versions of the plug- in are included: audio effect or virtual instrument. They both offer similar capabilities (you can write instruments in the effect version too for example), but they will be considered differently by most host applications.

Just use the synth version if your main purpose is to use virtual instruments scripts, and the effect version if you want to process the audio signal or MIDI events as an effect.

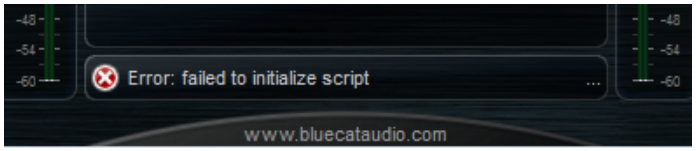
The Principle

Executing a Script

Once processing script has been selected, the plug- in will do the following:

1. Load the selected script file and its dependencies.
2. Compile the script into bytecode.
3. Compile the bytecode into machine code (JIT compiler).
4. Execute the compiled code to process audio or MIDI data in real time.

If an error occurs during any of these steps, it will be reported back to you at the bottom of the plug- in:



The complete error log can be loaded in a text editor by clicking on the error message above.

Warnings may also be reported. A script with warnings will run, but it is strongly recommended to fix them.

Writing a Script

The fastest way to get started is to select an existing script and modify it. The "Save As" feature was designed for this purpose.

The script defines variables (data) and functions that will be used by the plug- in to process incoming audio or MIDI data. The role of a script is to modify incoming data (audio or MIDI). The full interface between the plug- in and scripts is described later in this manual.

With Blue Cat's Plug'n Script, it is possible to many types of scripts, such as:

- Audio Processing script: modifies incoming audio data to apply an effect.
- MIDI Processing script: modifies incoming MIDI data to produce different events at the output of the plug- in.
- Synthesizer / virtual instrument: produces brand new audio data based on incoming MIDI events (usually ignores incoming audio data).
- Audio Generator: generates audio waveforms based on script parameters (usually ignores incoming audio and MIDI).
- MIDI events generator: generates new MIDI events based on script parameters (usually ignores incoming audio and MIDI).

Binary or Source / Angelscript or Native?

Starting with version 1.2, the plug- in supports two types of scripts:

"Source" or "text" scripts

These scripts use the angelscript engine: the source code is compiled on the fly by the plug- in using the JIT and executed inside the angelscript virtual machine. These scripts are:

- easier to write and debug, as they are crash- proof and errors are quickly detected by the virtual machine.
- fully portable: the same script can be loaded on all supported platforms (32 or 64- bit Mac and Windows)
- slower to execute than native code.

"Binary" or "native" scripts

Copyright (c) 2014-2015 Blue Cat Audio

The plug- in can also load scripts in the binary form, which are actually shared libraries (a.k.a. "dll") that have been externally compiled using a third party development environment.

These scripts:

- are faster than source scripts (the resulting plug- in is as fast as if it has been entirely written with native code from scratch).
- are usually more difficult to debug, as they are not executed in a sandbox: they have direct access to memory and may crash the software.
- let you use third party native libraries: there is no restriction and no possible conflict, as native dsp scripts are "standalone" shared libraries.
- can be distributed with or without source code: It is possible to distribute only the binary version, without the orginal source code, if you prefer not to disclose your algorithms. But the source can of course be shared with the community if you don't mind!
- have to be compiled separately for each platform (Mac, windows and 32 or 64- bit).

Our recommendation

If you are not familiar with programming, you should definitely start with angelscript.

Even if you are an experimented C + + developer and want to distribute a native library in the end, it is recommended to first write the plug- in as a text script, and then port it to C/ C + + once it has been fully validated:

- It is much faster to write scripts and reload them in the plug- in (no extra compilation step required).
- Finding errors is easier, and the script will not crash.

Once a script runs smoothly, porting it to C/ C + + is quite fast, as you can see in our porting guide.

Text Editor

To edit the source scripts, the plug- in does not provide a text editor. It is however easy to integrate with your favorite text editor with syntax highlighting by setting the appropriate application in the preferences, as described later in this manual.

If you do not already have a text editor that provides syntax highlighting, here are a few recommendations:

- Notepad 2 (Windows, free)
- Notepad + + (Windows, free)
- Visual C + + Express (Windows, free)
- XCode (Mac, free)
- Aquamacs (Mac, free)

IDE and Compiler

Binary scripts can be written and compiled using any third party IDE (Integrated Development Environment) capable of building a shared library that exports C functions. You can for example use XCode on Mac and Visual Studio on Windows.

Note: while not strictly required, a compiler compatible with the C + +11 standard is recommended, as it will simplify switching between angelscript and C + + code, as explained in details later in the manual.

Copyright (c) 2014-2015 Blue Cat Audio

The User Interface

Note: The main toolbar, menus and basic features available with all our plug- ins are detailed in the Blue Cat Audio Plug- ins Basics section.

Overview

The generic controls that can be found in all our plug- ins are not described in this chapter. More details for these are available in the next section.

The plug- in is delivered with a default generic skin that lets you load or edit scripts and displays the controls described in the script. It is also possible for you to create dedicated skins for a particular script if you wish. Additional specific skins for multiple scripts are already included in the plug- in and will not be described in details here.

1. Show/ hide the level meters.
2. Select large or narrow display. Changes the way controls are organized (single or two columns) when using more than 4 parameters for a script.
3. Reload and recompile the current script (can also be used to reset the script).
4. Open the script in the text editor specified in the preferences (not available for binary scripts).
5. Open the folder that contains the current script.
6. Open the preferences pane.
7. Current script. Click on the script name to open the scripts menu and select a script.
8. Information box: displays the name and description of the current script.
9. Plug- in controls and meters, as defined in the current script.
10. Input (peak) level meter.
11. Output (peak) level meter.
12. Status bar: displays the current status of the script engine and the last message. Click on the message to open the full log into the text editor selected in the preferences.

More Controls

Some scripts may also let you enter strings using text boxes, as shown below. Also, for parameters with discrete values, clicking on the value displayed below the name opens a popup menu for faster selection from a list:

Copyright (c) 2014-2015 Blue Cat Audio

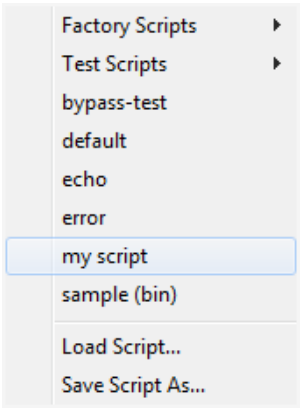
Script User Manual

Some scripts come with a user manual (in html or pdf format). In this case, an additional icon with a question mark will appear in the toolbar:

Click on the button to open the user manual for the current script.

The Script Menu

When clicking on the script selector (7), the following menu appears:



It shows a list of scripts that can be found in two locations: the factory scripts, provided by the plug- in (read- only), and your custom scripts, that are located in the documents folder. Just select one of the scripts listed here to load it into the plug- ins.

Note that for binary (native) scripts are listed as such. Scripts without the (bin) suffix are text scripts that can be edited.

Two additional commands are available at the bottom of the menu:

- **Load Script:** opens a file browser to load an external script file that has not been saved into the user documents sub- folder.
- **Save Script As:** copies the current script to another location and select it as current. Use this feature to reuse and modify an existing script (not available for binary scripts).

While it is recommended to keep the scripts in the documents folder (this lets you share plug- in presets with others easily), it may be useful to select existing scripts from other locations for testing purposes.

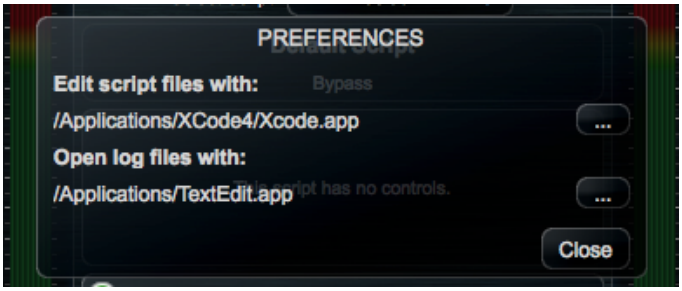
Meters

Copyright (c) 2014-2015 Blue Cat Audio

Some scripts may display read- only values (measurements, constants or user feedback). For this purpose, the default generic skin provides value meters that are displayed below parameters:

The Preferences Pane

In the preferences pane, you can select the applications that can be used to edit the text script files and open the error log from the user interface of the plug- in:



To change the application, click on its path or on the button at the right and select the appropriate application in the file browser.

The various elements of the user interface (knobs, sliders, buttons...) are simple and intuitive to operate, but more information about how to interact with them is available in the "Plug- ins Basics" chapter of this manual.

Operation

Using Existing Scripts

Even though the plug- in lets you write your own scripts, it is delivered with many factory scripts that you can use for music production or as a starting point for your own programming.

Loading an existing script is straightforward:

1. Click on the script selection drop down button (7)
2. Select the script in the list, or choose "Load Script" to open a file browser.

This will load and compile the script into the plug- in and show the controls and meters defined by the script: you can then interact with the script as if it were a standard plug- in. Automation and MIDI control are supported for all parameters just like with any other Blue Cat Audio plug- in.

Most factory scripts are delivered in both source and binary forms. For better performance, choose the binary version ("bin"), but if you want to modify the script, you will prefer the source version.

Organizing Scripts

Scripts used for the Plug'n Script plug- in are either text files using the *.cxx* extension or binary files (using the *.bin* extension on Mac and *.x64* or *.x86* on windows). We chose the cxx extension for source scripts because the syntax of the AngelScript language is very similar to C + + and most text editors will thus be able to perform syntax highlighting without additional configuration.

User scripts (as opposed to factory scripts) are located in a sub folder of the Documents directory:

[USER DOCUMENTS]/ Blue Cat Audio/ Blue Cat's Plug'n Script/ Scripts/

Scripts can be organized into sub folders, and the directory structure will be reproduced in the scripts menu to keep scripts organized.

Copyright (c) 2014-2015 Blue Cat Audio

Scripting Tutorial

As an introduction, in this chapter we will be writing text scripts using angelscript only. We will look at native versions in the next chapter. But the principles, classes and functions of the API (Application Programming Interface) remain the same for native scripts.

About Angelscript

Angelscript was originally designed for video games, with ease of use and performance in mind, which definitely suits our audio and MIDI processing needs.

From the AngelScript documentation: *The script language is based on the well known syntax of C + + and more modern languages such as Java, C#, and D. Anyone with some knowledge of those languages, or other script languages with similar syntax, such as Javascript and ActionScript, should feel right at home with AngelScript. Contrary to most script languages, AngelScript is a strongly typed language, which permits faster execution of the code and smoother interaction with the host application as there will be less need for runtime evaluation of the true type of values.*

This means that it is very easy to use the scripting language without really learning it. Compared to other scripting languages such as Javascript, it is also safer because it is strongly typed: less errors may occur during runtime, which is much better for our real time processing purposes. Last but not least, most of the code written with AngelScript can be quickly ported (if not just copied/ pasted) to C + +, if you need to reuse the algorithms in a native application or plug- in later: this is very useful for prototyping. By the way, switching back and forth between native and angelscript version of the same script is very easy, as you will see later.

Specific Features

One particularity with AngelScript though is the concept of handle, declared with a '@'. For our scripting purposes, you can just consider it as a pointer to an object. The only thing you need to remember here is that the pointer may be null (points to no object), so you need to check its value before using it:

```
// Get a handle to an object
object@ myObject=GetObject();

// check if handle points to an actual object (C++/Java style)
if(@myObject != null)
{
    // do something
}

// alternative syntax
if(myObject !is null)
{
    // do something
}
```

In the context of this plug- in, handles can be used for two purposes:

- Passing an optional object to a function (we use it for example for transport information, that may exist or not depending on the host application).
- Keeping a pointer to an object in order to avoid requesting it multiple times, for performance reasons (more on that later).

Also, like in C + +, it is possible to overload operators (adding behavior to objects for +,-,* etc.). The syntax is however different: for example, the *[] operator* (to access array elements) has to be defined as:

```
Object opIndex(uint i);
```

You can still access this operator using the usual *object[i]* syntax in the code calling the class, but in order to define or override this operator, you need to use the *opIndex* name. See the full list of operators for more information.

Another specific feature that is used in this plug- in is AngelScript's property accessors, which let you define setters and getters for properties of a class like in C#. The only thing you need to know here is that a class defined with get_length() method let you access it directly as a read only property named "length". If the set_length() method is defined too, the "length" property can be also modified.

You may want to read the Angelscript language reference for more details about the scripting language.

Copyright (c) 2014-2015 Blue Cat Audio

Scripts Types

For both native and angelscript, he dsp scripting framework proposes two types of scripts:

- **audio sample processing:** the script is called for every audio sample to process it. This is the most simple way to perform audio processing without the need to handle buffers and iterate thru samples. It is however limited: it cannot handle MIDI events, and transport information is not directly accessible in the processing routine. This type of script should implement the *processSample* function, as well as the *updateInputParameters*function that will be called to update internal variables from the input parameters array if any parameter is required.
- **block processing:** the *processBlock* function is called by the plug- in for every block of samples (buffer) provided by the host. You will want to use this type of script if specific parameters handling is required or if you want to process MIDI events. If no specific parameter smoothing is required, the *updateInputParametersForBlock* function will let you update parameters before every block, or you can use the start and end values provided in the *processBlock* call.

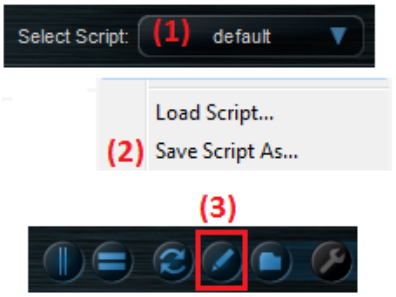
So depending on your needs and you capabilities you will have to choose one of these types of scripts.

Writing your first Audio Script

Let's start with the 'default' script that is loaded when first opening the plug- in. It does not do anything but declare its name:

```
string name="Default Script";
string description="Bypass";
```

In the scripts menu, click on "Save As..." and save the script with a new name. Then click on the "edit" button to open the script in your preferred editor as defined in the preferences:



As you can see there is not much in here! Let's add a simple processing function that just multiplies the current audio samples by a factor (.5, equivalent to -6 dB). Add the following lines in the script file:

```
void  processSample(array<double>& ioSample)
 {
   for(uint channel=0;channel<audioInputsCount;channel++)
   {
     ioSample[channel]*=.5;
  }
}
```

Save the file in the editor, and click on the refresh button in the plug- in:

As you can hear, the sound is now quieter by 6dB! You have written and loaded your first dsp script!

Adding parameters

Simple linear gain

Now, wouldn't it be nice to control the gain instead of having it fixed in the script? To do so, you can simply define a gain parameter in the script, as shown below. Since we would like to display the appropriate name in the user interface, we will define both the inputParameters array and the inputParametersNames array with a single element:

```
array<string> inputParametersNames={"Gain"};
array<double> inputParameters(inputParametersNames.length);
```


Copyright (c) 2014-2015 Blue Cat Audio

And use it in the processing function (the updated value is stored by the plug- in into the first element of the inputParameters array that we have defined):

```
void processSample(array<double>& ioSample)
{
  for(uint channel=0;channel<audioInputsCount;channel++)
  {
    ioSample[channel]*=inputParameters[0];
  }
}
```

By default, when no range is specified, parameter values are displayed as [0%-100%] but are passed to the script as values in the [0;1] range. So the script that you have just written will let you control the volume from 0 to 100%. If you refresh the script in the plug- in user interface, a new knob has appeared to let you control the gain:

Adding range, unit and more

Now in the real world, you usually want to use more complex parameters. So let's now change the script definition to accept a gain parameters in decibels (dB), ranging from -20 to +20 dB, with a default value of 0 dB:

```
array<string> inputParametersNames={"Gain"};
array<string> inputParametersUnits={"dB"};
array<double> inputParameters(inputParametersNames.length);
array<double> inputParametersMin={-20};
array<double> inputParametersMax={20};
array<double> inputParametersDefault={0};
```

We now have to transform incoming input parameter values (decibels) to a value that can be multiplied with audio samples). So we'll add an internal 'gain' variable and an implementation for the *updateInputParameters* function that updates this variable from our single input parameter. We will also modify the *processSample* function to use the 'gain' variable instead of the input parameters array:

```
/** Define our internal variables.
*
*/
double gain=0;

/** per-sample processing function: called for every sample with updated parameters values.
*
*/
void processSample(array<double>& ioSample)
{
  for(uint channel=0;channel<audioInputsCount;channel++)
  {
    ioSample[channel]*=gain;
  }
}
```


Copyright (c) 2014-2015 Blue Cat Audio

```
/** update internal parameters from inputParameters array.
*   called every sample before the processSample method when necessary.
*/
void updateInputParameters()
{
  /*using reverse dB formula: gain=10^(gaindB/20)*/
  gain=pow(10,inputParameters[0]/20);
}
```

Please note that the *updateInputParameters* function will be called for every sample while the parameter is being modified, to ensure continuous update of the gain and avoid audible steps. In the example above, we have also added comments (highlighted in green) to the code (the syntax is the same as C + + comments).

Reload the script in the plug- in. The new parameter values are displayed in the user interface, and right clicking on the knob sets it to 0 dB.

Using MIDI events

In order to use MIDI events, it is necessary to use the *processBlock* function. Let's restart a script from scratch that will filter MIDI events based on their type. You can now close your text editor window with your previous script.

Once again, let's load the 'default' factory script and select "Save As" in the script menu to save it under a new name. Click on the "edit" button to load it. We are now ready to go!

Since we'd like to use the MIDI utilities provided with the plug- in to work with MIDI events, let's include the MIDI utils header file. The path to this file is relative to the current file, so we suppose here that the script you are editing was saved in the root of the "Scripts" folder. If it was saved elsewhere, you may need to change this path:

```
#include "library/Midi.hxx"
```

In the processBlock function, we'll simply iterate on incoming MIDI events, and write to the output queue the MIDI events that we are interested in. Let's decide that we want to keep only Note On and Note Off events and drop others:

```
void processBlock(BlockData& data)
{
  for(uint i=0;i<data.inputMidiEvents.length;i++)
  {
    // send only Note On and Off events
    MidiEventType type=MidiEventUtils::getType(data.inputMidiEvents[i]);
    if(type==kMidiNoteOn || type==kMidiNoteOff)
    {
      // forward the event (unchanged)
      data.outputMidiEvents.push(data.inputMidiEvents[i]);
    }
  }
}
```

Reload the script in the plug- in: it will now pass to its MIDI output 'note on' and 'note off' events only and dismiss other types of events (in order to use the plug- in as a MIDI effect in your host application, you may need to perform specific routing. If you are an owner of the Blue Cat's PatchWork, you can for example use such a script to filter events sent to a synth loaded in a specific slot).

In this example we do not modify the events but simply filter them. Also, we do not worry about audio samples here, so they will be passed to the output of the plug- in, unchanged.

Audio and MIDI together: writing a simple synth

Using the *processBlock* function, you can use MIDI events to produce sound. Several examples of simple synthesizers are provided with the plug- in in the factory "Synth" scripts. You may want to start with the 'sin synth' script which is the most simple synthesizer you can write: it is monophonic and produces a simple sine wave. The 'sin synth 2' script adds simple dynamics (attack/ release) and pitch smoothing control.

Output Parameters

Copyright (c) 2014-2015 Blue Cat Audio

Like with some of our other plug- ins, you can write here scripts that generate output parameters that can be reused for writing automation or sending MIDI CC events to control other plug- ins. These output parameters can be defined in a similar way as input parameters:

```
array<string> outputParametersNames={"Out1","Out2"};
array<double> outputParameters(outputParametersNames.length);
```

They can be updated by implementing the *computeOutputData* function:

```
void computeOutputData()
{
   outputParameters[0]=// TODO...
   outputParameters[1]=// TODO...
}
```

You can for example check the 'level meter' script in the " Utility" folder for more details.

Input Strings

Starting with version 1.1, it is also possible to define input strings for dsp scripts. It can be used for example to load or save files from the script.

The 'wav file recorder' and 'wav file player' scripts show how to use this feature.

Side Chain input and Auxiliary Outputs

Starting with version 1.2, extra audio inputs (effect version) or outputs (synth version) may be available for side chain applications, or to build virtual instruments with auxiliary outputs. Note: these extra inputs and outputs are optional and may not be available or simply not connected in the host application.

The 'side chain a- b' script is an example using the side chain signal.

Next Steps

The plug- in is delivered with many factory scripts that show multiple ways of using the dsp script engine. The best way to get started with writing your own scripts is to load them and open them in the editor to read the code. These scripts embed many comments to help you understand how they were written.

Before modifying the factory scripts, use the "Save As" command in the scripts menu to save them in your own directory, like we did before with the 'default' script.

Once you are familiar with the API, you may also want to *go native* and port your scripts to native code to make them faster.

Sharing your Scripts and Presets

Once you have written a script you are happy with, you may want to share it with others. It is as easy as copying the script file (*and its dependencies if any*) to someone else's documents folder. If your script is using headers from the library, make sure that the relative to the library is the same on the destination machine. If it is not, the include directive in the script will have to be modified.

If you create presets that use custom scripts, please note that you also have to share your scripts together with the presets or they will fail to load.

To share your own scripts and presets on www.bluecataudio.com for end users, please contact us. We will be happy to upload your scripts and show them with a link in our download section.

We also encourage you to share your work with the developers community. We have setup a Plug'n Script code repository on GitHub to gather the scripts and skins. It is also a good place to show your knowledge and learn from others!

Tips and Tricks

Before we let you go write your own scripts, here are a few things for you to keep in mind:

- Scripts should be able to handle any audio I/ O configuration, or fail during initialization (use the *initialize* function that returns a boolean for this). You can see examples of such scripts in the Stereo category of the factory scripts.
- The 'library' folder contains some header files provided with the plug- in to help you write your own scripts. Do not modify these files or create your files in this folder, as they may be updated with the plug- in in the future.
- CPU usage may vary a lot depending on how you write your scripts. Once a script is functional, you may want to read the advanced programming section in this manual and optimize it.

Copyright (c) 2014-2015 Blue Cat Audio

You are now ready to dive into the scripting reference below and write your own scripts!

Scripting Reference

This chapter is about angelscript. Native scripts are covered later in this manual.

AngelScript Version

The plug- in currently uses AngelScript **2.29.2** together with the BlindMind Studios JIT compiler.

AngelScript Add- ons

The plug- in is using several additional modules provided with the AngelScript engine as script add- ons:

- Math functions (using double precision instead of float), with the addition of the exp() function.
- Script Arrays.
- Strings.
- File.

API

The programming interface (variables, functions and classes) defined by the plug- in is described below. All these elements are made available to scripts by the plug- in (angelscript version).

```
/*
*  PlugNScript Application Programming Interface
*  Functions and variables defined by the C++ scripting host
*  Do not include this file in your scripts: it is provided
*  for documentation purposes only. Classes and functions are already made available
*  to scripts by the plug-in.
*  Created by Blue Cat Audio <services@bluecataudio.com>
*  Copyright 2011-2015 Blue Cat Audio. All rights reserved.
*
*/

/// The number of audio inputs of the dsp filter
const uint audioInputsCount;
/// The number of audio outputs of the dsp filter
const uint audioOutputsCount;
/// The number of (optional) auxiliary audio inputs of the dsp filter
const uint auxAudioInputsCount;
/// The number of (optional) auxiliary audio outputs of the dsp filter
const uint auxAudioOutputsCount;

/// The maximum number of samples per block for block processing.
const int maxBlockSize;
/// The current sample rate.
const double sampleRate;

/// The path to the user documents folder on the file system
/// (Using '/' separators)
const string userDocumentsPath;

/// The path of the current dsp script file on the file system
/// (Using '/' separators)
const string scriptFilePath;

/** MIDI Event (packet) abstraction.
*Contains 4 bytes of data and a timestamp.
*/
class MidiEvent
{
   /// MIDI data (4 bytes packet)
   uint8 byte0;
```


Copyright (c) 2014-2015 Blue Cat Audio

```
    uint8 byte1;
    uint8 byte2;
    uint8 byte3;
    /// time stamp of the event, as an offset in samples
    /// from the beginning of the current block.
    double timeStamp;
};

/** List of MIDI Events.
*
*/
class MidiQueue
{
    /// Returns the number of events available in the queue.
    /// Accessed as "length" attribute.
    uint            get_length()const;
    /// Sets the number of events in the queue.
    /// Accessed as "length" attribute.
    void            set_length(uint l);
    /// [] operator: returns the Midi event located at index i in the queue.
    MidiEvent&      opIndex(uint i);
    /// [] operator: returns the Midi event located at index i in the queue.
    const MidiEvent& opIndex(uint i)const;
    /// pushes "evt" at the end of the events queue.
    void            push(const MidiEvent & evt);
};

/** Host Transport information.
*
*/
class TransportInfo
{
    /// Returns the current tempo (beats per minute)
    /// Can be accessed directly as "bpm" attribute.
    double  get_bpm()const;
    /// Returns the upper value of the time signature.
    /// Can be accessed directly as "timeSigTop" attribute.
    uint    get_timeSigTop()const;
    /// Returns the lower value of the time signature.
    /// Can be accessed directly as "timeSigBottom" attribute.
    uint    get_timeSigBottom()const;
    /// Returns true when the host application is playing.
    /// Can be accessed directly as "isPlaying" attribute.
    bool    get_isPlaying()const;
    /// Returns true when the transport of the host application is in a loop.
    /// Can be accessed directly as "isLooping" attribute.
    bool    get_isLooping()const;
    /// Returns true when the host application is recording.
    /// Can be accessed directly as "isRecording" attribute.
    bool    get_isRecording()const;
    /// Returns the position in samples of the first sample of the current
    /// buffer since the beginning of the song.
    /// Can be accessed directly as "positionInSamples" attribute.
    int64   get_positionInSamples()const;
    /// returns the position in quarter notes of the first sample of the
    /// current buffer since the beginning of the song.
    /// Can be accessed directly as "positionInQuarterNotes" attribute.
    double  get_positionInQuarterNotes()const;
    /// Returns the position in seconds of the first sample of the current
    /// buffer since the beginning of the song.
```


Copyright (c) 2014-2015 Blue Cat Audio

```
    /// Can be accessed directly as "positionInSeconds" attribute.
    double  get_positionInSeconds()const;
    /// Returns the position in quarter notes of the first bar of the current measure.
    /// Can be accessed directly as "currentMeasureDownBeat" attribute.
    double  get_currentMeasureDownBeat()const;
    ///  When looping, returns the position in quarter notes of the beginning of the loop.
    /// Can be accessed directly as "loopStart" attribute.
    double  get_loopStart()const;
    /// When looping, returns the position in quarter notes of the end of the loop.
    /// Can be accessed directly as "loopEnd" attribute.
    double  get_loopEnd()const;
};

/** Structure passed to the script for block processing.
*
*/
class BlockData
{
    /// An array containing audio buffers of each audio channel for this block.
    /// You can access sample i of channel ch using samples[ch][i].
    array< array<double> >@ samples;
    ///< The number of audio samples to process for this block.
    uint                    samplesToProcess;
    /// The incoming MIDI events queue.
    const MidiQueue@        inputMidiEvents;
    /// The MIDI events output queue to send MIDI events.
    MidiQueue@              outputMidiEvents;
    /// The input parameters values at the beginning of the block.
    const array<double>@    beginParamValues;
    /// The input parameters values at the ends of the block.
    const array<double>@    endParamValues;
    /// Transport information - may be null if not supported or provided
    /// by the host application.
    const TransportInfo@    transport;
};

/// Utility function that prints the content of a string to the log file.
void print(const string& in message);

/// Additional math function that produces a pseudo random number ranging
/// from min to max
double rand(double min=0,double max=1)
```

Script Reference

The interface used by the plug- in to communicate with the scripts is described below (angelscript version). All elements are optional. Just choose the variables and functions that you need to implement for a particular script.

Copyright (c) 2014-2015 Blue Cat Audio

```
/**
*  Blue Cat's PlugNScript Scripting Reference.
*  Shows functions and attributes that can be defined by the dsp script.
*  Do not include this file in your scripts: it is provided only
*  for documentation purposes.
*  You can however use it as a starting point to write your own scripts.
*  All methods and attributes are optional and can be ignored.
*  You can for example write a script with just a single function defined.
*  Created by Blue Cat Audio <services@bluecataudio.com>
*  Copyright 2011-2015 Blue Cat Audio. All rights reserved.
*
*/

// Script metadata----------------------------------------
/// The name of the script to be displayed in the plug-in.
string name="Script Name";
/// The short description of the script to be displayed in the plug-in.
string description="Script Description";
/// The author of the script to be displayed in the plug-in.
string author="Script Author";

// Script I/O parameters and strings and associated metadata----------
/// An array of parameters to be used as input for the script.
/// Will be displayed in the user interface of the plug-in and accessible for
/// automation and MIDI control.
array<double> inputParameters(2);
/// Names to be displayed in the plug-in for the input parameters.
array<string> inputParametersNames={"P1","P2"};
/// Units for the corresponding input parameters.
array<string> inputParametersUnits={"dB","%"};
/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
/// for non-enum parameters. Require that min and max values are defined.
array<string> inputParametersEnums={"value1;value2",""};
/// Minimum values for the corresponding input parameters. Default value is 0.
array<double> inputParametersMin={0, 0};
/// Maximum values for the corresponding input parameters. Default value is 1.
array<double> inputParametersMax={10,20};
/// Default values for the corresponding input parameters. Default value is 0.
array<double> inputParametersDefault={5,0};
/// Number of steps for the corresponding input parameters.
/// Default is -1 (continuous control - no steps).
array<int>    inputParametersSteps={10,20};

/// An array of strings to be used as input for the script.
/// Will be displayed in the user interface of the plug-in
array<string> inputStrings(2);
/// Names to be displayed in the plug-in for the input strings.
array<string> inputStringsNames={"S1","S2"};

/// An array of parameters to be used as output of the script.
/// Will be displayed in the user interface of the plug-in as meters and accessible
/// to generate automation and MIDI controllers.
array<double> outputParameters(2);
/// Names to be displayed in the plug-in for the output parameters.
array<string> outputParametersNames={"OUT 1"," OUT 2"};
/// Units for the corresponding input parameters.
array<string> outputParametersUnits={"dB","dB"};
/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
```


Copyright (c) 2014-2015 Blue Cat Audio

```
/// for non-enum parameters. Require that min and max values are defined.
array<string> outputParametersEnums={"value1;value2",""};
/// Minimum values for the corresponding input parameters. Default value is 0.
array<double> outputParametersMin={0, 0};
/// Maximum values for the corresponding input parameters. Default value is 1.
array<double> outputParametersMax={10,20};
/// Default values for the corresponding input parameters. Default value is 0.
array<double> outputParametersDefault={5,0};

/** Initialization: called right after the script has been compiled
*   and before any other processing occurs.
*   Ignored if the initialize function returning a boolean is defined.
*/
void initialize()
{
}

/** Initialization: called right after the script has been compiled
*   and before any other processing occurs.
*   return false if initialization fails (for example if the number of
*   audio channels or the sample rate are not compatible).
*   When returning false, it is strongly advised to print a message with
*   the "print" function for the end user using the print function.
*/
bool initialize()
{
   /// write script initialization here
   /// and return false if something is not supported (number of channels, sample rate...)
   return true;
}

/** Reset the state of the filter.
*
*/
void reset()
{
}

/** Returns the tail size in samples.
*   use -1 for infinite (typically for audio or MIDI generators or synths).
*   use 0 if the processor does not produce any sound when fed with silence (default).
*/
int getTailSize()
{
   return 0;
}

/** Returns the latency added by the script if any, in samples.
*   Returns 0 by default.
*/
int getLatency()
{
   return 0;
}

/** Per-sample processing function: called for every sample with updated parameters values.
*   If defined, the processBlock function is ignored.
*   ioSample: an array of current audio samples (one element for each audio channel).
* You can access the current sample of channel 'ch' using ioSample[ch].
*/
```


Copyright (c) 2014-2015 Blue Cat Audio

```
void processSample(array<double>& ioSample)
{
}

/** Update internal parameters from inputParameters array when required.
*   Called every sample, right before the processSample() method, or every block,
*   before the processBlock() method.
*   This function will not be called if input parameters have not been modified since last call.
*/
void updateInputParameters()
{
}

/** Per-block processing function: called for every block with updated parameters values.
*   Ignored if the processSample() function is defined.
*/
void processBlock(BlockData& data)
{
}

/** Called for every block to update internal parameters from the inputParameters and
*   inputStrings arrays that have been updated by the host.
*   This function will not be called if input parameters have not been modified since last call,
*   and if transport info (tempo and time signature only) are unchanged.
*/
void updateInputParametersForBlock(const TransportInfo@ info)
{
}

/** Update output parameters values array from internal variables.
*
*/
void computeOutputData()
{
}
```

Advanced Scripting Topics

This chapter described advanced programming topics for angelscript dsp scripts. Native programming is cover in the native programing chapters.

Debugging Scripts

The plug- in does not have an included debugger for scripts. Since the scripts are usually called in a real- time audio thread, this would not be very useful anyway. So here are several alternatives to help you debug your scripts:

Use the print Function

You can quickly dump the content of your script variables with the print function:

```
// script data
double myData;

// ...
// dump the content of data to the log
print("myData=" + myData);
```

It will be appended to the log file and will be made visible in the user interface of the plug- in (one line displayed at the bottom). However, please be aware that if you call this function in the main processing loop, it may quickly create a huge log file.

Use Output Parameters

Copyright (c) 2014-2015 Blue Cat Audio

For variables that vary a lot over time and require regular updates, you can dump them to output parameters. These parameters will be displayed in the user interface of the plug- in. They can also be recorded as automation lanes in host applications if you want to display them over time.

Use the MIDI Log script

When writing MIDI plug- ins, it can be useful to use the included MIDI Log script as a target plug- in to check the MIDI events produced by your own script in human- readable format. Recording MIDI tracks in a sequencer may help too!

Optimization and Best Practices

The main part of the scripts that is worth optimizing is the portion that is called for every audio sample. This means that you will want to be more cautious about the following pieces:

- The entire *processSample* function (called every sample).
- The entire *updateInputParameters* function (called every sample).
- The loops on audio samples in the *processBlock* function.

Everything else does not have a large influence on CPU usage, so there is no need to spend time on optimizing the scripts there.

Despite the JIT compiler and strong typing, since AngelScript is a sandboxed scripting language, the engine cannot optimize the code in the same way as a C or C + + compiler would do. So here are a few tips to help you optimize your scripts:

- **Function Calls:** calls to custom script functions cannot be inlined and can be costly. So it is recommended to minimize function calls inside the sample processing loops.
- **Arrays:** calling the [] operator on an array to access elements is equivalent to a function call and cannot be optimized because of boundaries checking. So you should try to access data in arrays just once and keep the value (or a handle to the value) in a local variable to manipulate it.
- **Memory Allocation:** In the AngelScript engine, all objects except simple types (such as integers, floats, doubles etc.) are dynamically allocated: there is no stack like in C/ C + +. If you need variables for temporary data, you should instantiate them at the beginning of the script and avoid creating them in functions called by the plug- in. This avoids allocating memory in time critical tasks, which could cause drop outs.

Copyright (c) 2014-2015 Blue Cat Audio

Native Programming Tutorial

In this chapter we will assume that you have already read the Scripting Tutorial and that you know how to create dsp scripts with angelscript. Writing native scripts is not much more complicated but require a bit more programming knowledge and experience.

About Native (Binary) Scripts

Native or *binary* scripts are *compiled* dynamic libraries that export a simple C interface which is very similar to the angelscript interface., using global variables and functions

The binaries have to be compiled specifically for each platform (Mac or Windows - 32 or 64- bit), but the source code can be cross-platform.

Native Scripts on Mac

On **Mac**, native scripts should be compiled as **dynamically loaded shared libraries ('mh_bundle')**, with the *bin* extension. It is recommended to build them as 32/64- bit universal binary, so that they can be loaded by the plug- in in both 32 and 64- bit applications.

Native Scripts on Windows

On **Windows**, native scripts should be compiled as **dlls**, with the *x64* extension for the 64- bit version and .x86 for the 32- bit version. Thanks to this naming convention, when switching between 32 and 64- bit applications, presets and sessions will be restored accordingly, using the appropriate version of the binary.

Mac/ Windows compatibility

In order to ensure presets and sessions compatibility, the same script should be compiled with the appropriate extension (*bin*,*x64* or *x86*) and the same name on Windows on Mac.

API

Except from a few differences (see next chapter), the dsp API is very similar to the angelscript version. Helper classes are provided so that if you use a C + +11 compatible compiler, existing text scripts written with angelscript should almost compile directly (see the porting guide below).

The principle is the same as for text scripts: you only define the global variables and functions of the API that you need, and export their symbols (using the DSP_EXPORT macro). For example, the gain sample:

```
// C++ scripting support----------------------------
#include "dspapi.h"
#include "cpphelpers.h"

DSP_EXPORT uint    audioInputsCount=0;

/** \file
*   Apply selected gain to audio input.
*/

DSP_EXPORT string description="simple volume control";

DSP_EXPORT array<string> inputParametersNames={"Gain"};
DSP_EXPORT array<double> inputParameters(inputParametersNames.length);

double gain=0;

DSP_EXPORT void processSample(double ioSample[])
{
  for(uint channel=0;channel<audioInputsCount;channel++)
  {
    ioSample[channel]*=gain;
  }
}

DSP_EXPORT void updateInputParameters()
```


Copyright (c) 2014-2015 Blue Cat Audio

```
{
  gain=inputParameters[0];
}
```

Note: global variables are not shared among several instances of the same binary script, as each instance of the script is loaded into a separate memory space. If you want to share data among instances, you can use another dynamically loaded library that you load in the initialize function of the script.

Typical Sequence

1. The plug- in loads the shared library (binary script) in its own memory space: global variables are created.
2. The plug- in initializes global variables.
3. The plug- in calls the initialize function.
4. The plug- in calls other functions during processing and updates the values of the parameters.
5. The plug- in calls the shutdown function.
6. The plug- in unloads the shared library.

Differences with Angelscript

Variables definition and initialization

In angelscript, the host- side global variables of the API such as audioInputsCount, sampleRate, audioOutputCount etc. are defined and initialized by the host prior to execution, so they can be used directly when initializing the variables of the script. For native code, it is a bit different:

1. variables expected by the script have to be defined and exported with the DSP_EXPORT macro. They act as a placeholder in the code that will be filled by the host.
2. these variables remain uninitialized until the *initialize* function is called.

It means that the other variables and data structures that depend on the number of audio inputs, outputs or the sample rate have to be initialized in the initialize function. They can not be initialized with a global constructor like with angelscript.

Symbols export

With native scripts, public symbols (variables and functions of the API) have to be exported explicitely, with C- style naming. the DSP_EXPORT macro is provided for this purpose:

```
DSP_EXPORT string const name="Script Name";
DSP_EXPORT uint audioInputsCount=0;
DSP_EXPORT void processBlock(BlockData& data){}
```

Strings

The API uses only non modifiable C- style strings (zero terminated char*), and the api header defines "string" as char const* so that when using the C + + helpers, existing scripts will compile.

You may however use the std::string class in your scripts (or your own class) instead of old- style C- strings. you just need to make sure that the C- strings made available to the host are kept alive as long as necessary. Several sample scripts show dynamically allocated strings and how they can be handled.

The initialize function

In the native version, only a single initialize function can be defined, and it returns a boolean value, whereas in angelscript you have the choice to define the function with or without a return value.

```
DSP_EXPORT  bool  initialize()
```

The shutdown function

In order to be able to deallocate resources (free memory, close files etc.), a shutdown function is provided. In most cases it is not used (especially if you use C + + with destructors), but when using a straight- C implementation, you will probably need to implement it:

```
DSP_EXPORT  void  shutdown()
```

The processSample function

The signature of the processSample function has been changed to the following:

Copyright (c) 2014-2015 Blue Cat Audio

```
DSP_EXPORT void processSample(double ioSample[])
```

Beware that if you are using the C + + array helper, the angelscript signature using an array<double>& will compile properly, but the script will crash at runtime.

Getting Started: Samples

Most factory scripts are available as both angelscript and native processors, with their source code.

For native processors, an XCode project and a Visual Studio solution are provided. You can check their README file for more details about these projects and their structure. You can use them as a starting point for your own native scripts. They are already configured with appropriate build options.

On Mac you can simply duplicate a target to create your own script, and on Windows, just copy and modify an existing Visual Studio project.

Angelscript to Native DSP Porting Guide

Overview

Porting an existing script to native code is as simple as following these steps:

1. Copy the cxx file and rename it to cpp or cc.
2. Create a copy of any sample project and rename it.
3. In the new project, remove the old cpp file and replace it with yours.
4. Open the cpp file for edition and follow the porting check list below.

Porting check list

- Add api header includes at the top of the file (dspapi.h and cpphelpers.h).
- Rename existing .hxx library includes to .h.
- Add the declaration of the symbols used by the script, for example "DSP_EXPORT double sampleRate=0;"
- Add the **DSP_EXPORT** directive to all symbols that should be exported.
- Initialize global variables appropriately, using the *DSP_EXPORT bool initialize()* for variables that depend on globals.
- Change the *void initialize()* function definition to *DSP_EXPORT bool initialize()* if the void version was used.
- Replace @ with *, and calls to handles (.) with calls to pointers (- >)
- Strings: either use std::string on your own string class. You can use std::to_string (C + +11) to append int or double values to strings.
- Change the signature of the *processSample* function if used: change *array<double>@* to *double**.
- Implement the *shutdown* function to deallocate resources if necessary.
- You may have to add functions definitions before using them (angelscript does not require it).
- Change overloaded operators to their appropriate C + + counterpart (for example, *opEquals* should be changed to *operator* ==)
- Change *class* to *struct* or make data/ methods public as it is not the default in C + + as opposed to angelscript.

Troubleshooting

Common issues and their solutions:

- Parameters do no show? Check that parameters array is exposed using the DSP_EXPORT macro, and check variables names.
- Functions not called? Check functions naming and DSP_EXPORT directive.
- Crash when accessing ioSample[]? Change the processSample function signature using double* instead of array<double>.
- Missing symbols when using print function? Check that the host and hostPrint symbols have been declared in the script as follows: DSP_EXPORT void* host=null; DSP_EXPORT HostPrintFunc* hostPrint=null;
- Script compiles fine but fails when loaded? Check that *initialize* returns a boolean value (not void).

Debugging

To debug the native scripts, you can attach the debugger to the process hosting the Plug' n Script plug- in and set break points in the script. It is not different from debugging a standard shared library.

Native Programming Reference

API

Copyright (c) 2014-2015 Blue Cat Audio

The basic data structures for the API are defined in the include/ dspapi.h header file (See the code repository). It is compatible with most C and C + + compilers.

There is no dependency on extra libraies, just a header file defining C structures layout. When used with C + + extra inline methods are provided for compatibility with existing angelscript code. Also, typedefs are provided for common integer and floating point numbers used in angelscript.

```
/** Blue Cat's Plug'n Script dsp api header file.
 *  Contains all definitions required for DSP scripting in C or C++
 */

#ifndef _dspapi_h
#define _dspapi_h

// handling "bool" definition for C compilers
#ifndef __cplusplus
#if(!defined(_MSC_VER) || _MSC_VER >= 1900)
#include <stdbool.h>  // using the C99 definition of bool
 #else
#define bool char  // for binary compatibility with C++
 #define true 1
#define false 0
#endif
#endif

// Visual Studio does not support the inline keyword in C before VS2015
#if(!defined(_MSC_VER) || _MSCVER>=1900 || defined(__cplusplus))
#define INLINE_API inline
#else
#define INLINE_API
#endif

// basic types definitions for angelscript compatibility----------------
typedef  char const*     string;
typedef  unsigned int    uint;
typedef  signed char     int8;
typedef  unsigned char   uint8;
#ifdef _MSC_VER  // Visual Studio
 typedef _int64          int64;
typedef  unsigned _int64 uint64;
#else  // GCC compatible compiler assumed
 #include <stdint.h>
typedef  int64_t         int64;
typedef  uint64_t        uint64;
#endif

#define null 0

/** MIDI Event (packet) abstraction.
 *Contains 4 bytes of data and a timestamp.
 */
struct MidiEvent
{
   /// MIDI data (4 bytes packet)
  uint8 byte0;
  uint8 byte1;
  uint8 byte2;
  uint8 byte3;
   /// time stamp of the event, as an offset in samples
   /// from the beginning of the current block.
```


Copyright (c) 2014-2015 Blue Cat Audio

```
    double timeStamp;


#ifdef __cplusplus  // a default costructor is provided for C++
     MidiEvent():byte0(0),byte1(0),byte2(0),byte3(0),timeStamp(0){}
#endif
};

struct MidiQueue;

/// push function type definition for C implementation
typedef  void  (MidiQueuePushEventFunc)(struct MidiQueue* queue,const struct MidiEvent* evt);

/** List of MIDI Events.
 *
 */
struct MidiQueue
{
    /// Midi Events array
    struct MidiEvent* events;

    /// The number of events available in the queue. Never change this value
    /// Adding events should only be done using the fPush function (C) or push method (C++)
   uint            length;

    /// pushes an event at the end of the events queue.
   MidiQueuePushEventFunc*   pushEvent;


    // extra C++ inline methods for angelscript compatibility
#ifdef __cplusplus
    /// pushes "evt" at the end of the events queue.
    inline  void  push(const MidiEvent  & evt)
    {
       if(pushEvent!=null)
          pushEvent(this,&evt);
    }

    /// random access operator to access events directly (angelscript compatibility)
    inline  const MidiEvent&  operator  [](uint i)const
    {
       return events[i];
    }

    /// random access operator to access events directly (angelscript compatibility)
    inline MidiEvent&  operator  [](uint i)
    {
       return events[i];
    }
#endif
};

/** Host Transport information.
 *
 */
struct TransportInfo
{
    /// Returns the current tempo (beats per minute)
    /// Can be accessed directly as "bpm" attribute.
    double  bpm;
```


Copyright (c) 2014-2015 Blue Cat Audio

```
    /// Returns the upper value of the time signature.
    /// Can be accessed directly as "timeSigTop" attribute.
    uint    timeSigTop;
    /// Returns the lower value of the time signature.
    /// Can be accessed directly as "timeSigBottom" attribute.
    uint    timeSigBottom;
    /// Returns true when the host application is playing.
    /// Can be accessed directly as "isPlaying" attribute.
    bool    isPlaying;
    /// Returns true when the transport of the host application is in a loop.
    /// Can be accessed directly as "isLooping" attribute.
    bool    isLooping;
    /// Returns true when the host application is recording.
    /// Can be accessed directly as "isRecording" attribute.
    bool    isRecording;
    /// Returns the position in samples of the first sample of the current
    /// buffer since the beginning of the song.
    /// Can be accessed directly as "positionInSamples" attribute.
    int64   positionInSamples;
    /// returns the position in quarter notes of the first sample of the
    /// current buffer since the beginning of the song.
    /// Can be accessed directly as "positionInQuarterNotes" attribute.
    double  positionInQuarterNotes;
    /// Returns the position in seconds of the first sample of the current
    /// buffer since the beginning of the song.
    /// Can be accessed directly as "positionInSeconds" attribute.
    double  positionInSeconds;
    /// Returns the position in quarter notes of the first bar of the current measure.
    /// Can be accessed directly as "currentMeasureDownBeat" attribute.
    double  currentMeasureDownBeat;
    ///  When looping, returns the position in quarter notes of the beginning of the loop.
    /// Can be accessed directly as "loopStart" attribute.
    double  loopStart;
    /// When looping, returns the position in quarter notes of the end of the loop.
    /// Can be accessed directly as "loopEnd" attribute.
    double  loopEnd;
};

#ifdef __cplusplus
#define MidiQueueRef MidiQueue&
#else
#define MidiQueueRef MidiQueue*
#endif

/** Structure passed to the script for block processing.
 *
 */
struct BlockData
{
    /// An array containing audio buffers of each audio channel for this block.
    /// You can access sample i of channel ch using samples[ch][i].
    double**                samples;
    /// The number of audio samples to process for this block.
    uint                    samplesToProcess;
    /// The incoming MIDI events queue.
    const struct MidiQueueRef   inputMidiEvents;
    /// The MIDI events output queue to send MIDI events.
    struct MidiQueueRef         outputMidiEvents;
    /// The input parameters values at the beginning of the block.
    const double*               beginParamValues;
```


Copyright (c) 2014-2015 Blue Cat Audio

```
    /// The input parameters values at the ends of the block.
    const double*               endParamValues;
    /// Transport information - may be null if not supported or provided
    /// by the host application.
    const struct TransportInfo* transport;
};

// C API definition
#ifdef __cplusplus
#define EXTERN_C extern  "C"
#else
#define EXTERN_C
#endif

#ifdef _MSC_VER
#define DSP_EXPORT EXTERN_C __declspec(dllexport)
#else
#define DSP_EXPORT EXTERN_C __attribute__ ((visibility("default")))
#endif

typedef  void  (HostPrintFunc)(void* hostImpl,const char* message);

DSP_EXPORT  void*         host;
DSP_EXPORT HostPrintFunc* hostPrint;

// for angelscript compatibility
static INLINE_API  void  print(const char* message)
{
    // be paranoid
    if(hostPrint!=null  && host!=null)
        hostPrint(host,message);
}

#endif
```

C + + Script Reference

A C + +11- compatible script API is described below, as defined in the script_reference.cpp sample (See the code repository).

if you are using an older C + + compiler that does not support the new C + +11 initializer lists, compatible data structures can be defined in two steps instead of a single one, as shown for the inputParametersNames variable below.

The cpphelpers header file provides C + + helper classes that let you define data structures and functions with a syntax that is very similar to the angelscript version. If you prefer straight- C implementation, you can use the C definition instead.

Copyright (c) 2014-2015 Blue Cat Audio

```
#include "dspapi.h"
#include "cpphelpers.h"

// expected from host (Warning: appropriate values are filled before calling
// the initialize() function - do not use before)
DSP_EXPORT uint    audioInputsCount=0;
DSP_EXPORT uint    audioOutputsCount=0;
DSP_EXPORT uint    auxAudioInputsCount=0;
DSP_EXPORT uint    auxAudioOutputsCount=0;
DSP_EXPORT  int    maxBlockSize=0;
DSP_EXPORT  double  sampleRate=0;
DSP_EXPORT string  userDocumentsPath=null;
DSP_EXPORT string  scriptFilePath=null;
DSP_EXPORT  void*   host=null;
DSP_EXPORT HostPrintFunc* hostPrint=null;

// Script metadata----------------------------------------
/// The name of the script to be displayed in the plug-in.
DSP_EXPORT string  const name="Script Name";
/// The short description of the script to be displayed in the plug-in.
DSP_EXPORT string description="Script Description";

// Script I/O parameters and strings and associated metadata----------
/// An array of parameters to be used as input for the script.
/// Will be displayed in the user interface of the plug-in and accessible for
/// automation and MIDI control.
DSP_EXPORT array<double>  inputParameters(2);
/// Names to be displayed in the plug-in for the input parameters.
string names[]={"P1","P2"};
DSP_EXPORT array<string>  inputParametersNames(names);
/// Units for the corresponding input parameters.
DSP_EXPORT array<string> inputParametersUnits{"dB","%"};
/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
/// for non-enum parameters. Require that min and max values are defined.
DSP_EXPORT array<string> inputParametersEnums={"value1;value2",""};
/// Minimum values for the corresponding input parameters. Default value is 0.
DSP_EXPORT array<double> inputParametersMin={0,  0};
/// Maximum values for the corresponding input parameters. Default value is 1.
DSP_EXPORT array<double> inputParametersMax={10,20};
/// Default values for the corresponding input parameters. Default value is 0.
DSP_EXPORT array<double> inputParametersDefault={5,0};
/// Number of steps for the corresponding input parameters.
/// Default is -1 (continuous control - no steps).
DSP_EXPORT array<int>    inputParametersSteps={10,20};

/// An array of strings to be used as input for the script.
/// Will be displayed in the user interface of the plug-in
DSP_EXPORT array<string>  inputStrings(2);
/// Names to be displayed in the plug-in for the input strings.
DSP_EXPORT array<string> inputStringsNames={"S1","S2"};

/// An array of parameters to be used as output of the script.
/// Will be displayed in the user interface of the plug-in as meters and accessible
/// to generate automation and MIDI controllers.
DSP_EXPORT array<double>  outputParameters(2);
/// Names to be displayed in the plug-in for the output parameters.
DSP_EXPORT array<string> outputParametersNames={"OUT 1"," OUT 2"};
/// Units for the corresponding input parameters.
DSP_EXPORT array<string> outputParametersUnits={"dB","dB"};
```


Copyright (c) 2014-2015 Blue Cat Audio

```
/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
/// for non-enum parameters. Require that min and max values are defined.
DSP_EXPORT array<string> outputParametersEnums={"value1;value2",""};
/// Minimum values for the corresponding input parameters. Default value is 0.
DSP_EXPORT array<double> outputParametersMin={0,  0};
/// Maximum values for the corresponding input parameters. Default value is 1.
DSP_EXPORT array<double> outputParametersMax={10,20};
/// Default values for the corresponding input parameters. Default value is 0.
DSP_EXPORT array<double> outputParametersDefault={5,0};

/** Initialization: called right after the script has been compiled
*   and before any other processing occurs.
*   return false if initialization fails (for example if the number of
*   audio channels or the sample rate are not compatible).
*   When returning false, it is strongly advised to print a message with
*   the "print" function for the end user using the print function.
*/
DSP_EXPORT  bool  initialize()
{
   /// write script initialization here
   /// and return false if something is not supported (number of channels, sample rate...)
   return true;
}

/** Reset the state of the filter.
*
*/
DSP_EXPORT  void  reset()
{
}

/** Returns the tail size in samples.
*   use -1 for infinite (typically for audio or MIDI generators or synths).
*   use 0 if the processor does not produce any sound when fed with silence (default).
*/
DSP_EXPORT  int  getTailSize()
{
   return  0;
}

/** Returns the latency added by the script if any, in samples.
*   Returns 0 by default.
*/
DSP_EXPORT  int  getLatency()
{
   return  0;
}

/** Per-sample processing function: called for every sample with updated parameters values.
*   If defined, the processBlock function is ignored.
*   ioSample: an array of current audio samples (one element for each audio channel).
* You can access the current sample of channel 'ch' using ioSample[ch].
*/
/*DSP_EXPORT void processSample(double ioSample[])
{

}*/

/** Update internal parameters from inputParameters array when required.
```


Copyright (c) 2014-2015 Blue Cat Audio

```
*   Called every sample, right before the processSample() method, or every block,
*   before the processBlock() method.
*   This function will not be called if input parameters have not been modified since last call.
*/
DSP_EXPORT  void  updateInputParameters()
{
}

/** Per-block processing function: called for every block with updated parameters values.
*   Ignored if the processSample() function is defined.
*/
DSP_EXPORT  void  processBlock(BlockData& data)
{

}

/** Called for every block to update internal parameters from the inputParameters and
*   inputStrings arrays that have been updated by the host.
*   This function will not be called if input parameters have not been modified since last call,
*   and if transport info (tempo and time signature only) are unchanged.
*/
DSP_EXPORT  void  updateInputParametersForBlock(const TransportInfo* info)
{
}

/** Update output parameters values array from internal variables.
*
*/
DSP_EXPORT  void  computeOutputData()
{
}

/** Cleanup on shutdown: release all resources (allocated memory, files, system handles...).
 *
 */
DSP_EXPORT  void  shutdown()
{

}
```

C Script Reference

In order to build native scripts in straight C, the following API definition can be used, as shown in the script_reference.c sample. The provided chelpers header file define the simple C structures that are expected by the host.

```
#include  "dspapi.h"
#include  "chelpers.h"

// expected from host (Warning: appropriate values are filled before calling
// the initialize() function - do not use before)
DSP_EXPORT uint    audioInputsCount=0;
DSP_EXPORT uint    audioOutputsCount=0;
DSP_EXPORT uint    auxAudioInputsCount=0;
DSP_EXPORT uint    auxAudioOutputsCount=0;
DSP_EXPORT  int    maxBlockSize=0;
DSP_EXPORT  double  sampleRate=0;
DSP_EXPORT string  userDocumentsPath=null;
DSP_EXPORT string  scriptFilePath=null;
DSP_EXPORT  void*   host=null;
DSP_EXPORT HostPrintFunc* hostPrint=null;
```


Copyright (c) 2014-2015 Blue Cat Audio

```
// Script metadata----------------------------------------
/// The name of the script to be displayed in the plug-in.
DSP_EXPORT string  const name="Script Name";
/// The short description of the script to be displayed in the plug-in.
DSP_EXPORT string description="Script Description";

// Script I/O parameters and strings and associated metadata----------
/// An array of parameters to be used as input for the script.
/// Will be displayed in the user interface of the plug-in and accessible for
/// automation and MIDI control.
double _inputParameters[2];
DSP_EXPORT  struct CDoubleArray inputParameters={_inputParameters,2};

/// Names to be displayed in the plug-in for the input parameters.
const char* _inputParametersNames[]={"P1","P2"};
DSP_EXPORT  struct CStringArray inputParametersNames={_inputParametersNames,2};

/// Units for the corresponding input parameters.
const char* _inputParametersUnits[]={"%","dB"};
DSP_EXPORT  struct CStringArray inputParametersUnits={_inputParametersUnits,2};

/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
/// for non-enum parameters. Require that min and max values are defined.
const char* _inputParametersEnums[]={"value1;value2",""};
DSP_EXPORT  struct CStringArray inputParametersEnums={_inputParametersEnums,2};

/// Minimum values for the corresponding input parameters. Default value is 0.
double _inputParametersMin[]={0,0};
DSP_EXPORT  struct CDoubleArray inputParametersMin={_inputParametersMin,  2};

/// Maximum values for the corresponding input parameters. Default value is 1.
double _inputParametersMax[]={10,20};
DSP_EXPORT  struct CDoubleArray inputParametersMax={_inputParametersMax,  2};

/// Default values for the corresponding input parameters. Default value is 0.
double _inputParametersDefault[]={5,0};
DSP_EXPORT  struct CDoubleArray inputParametersDefault={_inputParametersDefault,  2};

/// Number of steps for the corresponding input parameters.
/// Default is -1 (continuous control - no steps).
int _inputParametersSteps[]={5,0};
DSP_EXPORT  struct CIntArray inputParametersSteps={_inputParametersSteps,  2};

/// An array of strings to be used as input for the script.
/// Will be displayed in the user interface of the plug-in
const char* _inputStrings[]={"",""};
DSP_EXPORT  struct CStringArray inputStrings={_inputStrings,2};

/// Names to be displayed in the plug-in for the input strings.
const char* _inputStringsNames[]={"S1","S2"};
DSP_EXPORT  struct CStringArray inputStringsNames={_inputStringsNames,2};

/// An array of parameters to be used as output of the script.
/// Will be displayed in the user interface of the plug-in as meters and accessible
/// to generate automation and MIDI controllers.
double _outputParameters[]={0,0};
DSP_EXPORT  struct CDoubleArray outputParameters={_outputParameters,  2};

/// Names to be displayed in the plug-in for the output parameters.
```


Copyright (c) 2014-2015 Blue Cat Audio

```
const char* _outputParametersNames[]={"OUT 1"," OUT 2"};
DSP_EXPORT  struct CStringArray outputParametersNames={_outputParametersNames,2};

/// Units for the corresponding input parameters.
const char* _outputParametersUnits[]={"dB"," dB"};
DSP_EXPORT  struct CStringArray outputParametersUnits={_outputParametersUnits,2};

/// Enumeration values for the corresponding input parameters.
/// Array of strings containing semicolon separated values. Use empty strings
/// for non-enum parameters. Require that min and max values are defined.
const char* _outputParametersEnums[]={"value1;value2",""};
DSP_EXPORT  struct CStringArray outputParametersEnums={_outputParametersEnums,2};

/// Minimum values for the corresponding input parameters. Default value is 0.
double _outputParametersMin[]={0,0};
DSP_EXPORT  struct CDoubleArray outputParametersMin={_outputParametersMin,  2};

/// Maximum values for the corresponding input parameters. Default value is 1.
double _outputParametersMax[]={10,20};
DSP_EXPORT  struct CDoubleArray outputParametersMax={_outputParametersMax,  2};

/// Default values for the corresponding input parameters. Default value is 0.
double _outputParametersDefault[]={5,0};
DSP_EXPORT  struct CDoubleArray outputParametersDefault={_outputParametersDefault,  2};


/** Initialization: called right after the script has been compiled
*   and before any other processing occurs.
*   return false if initialization fails (for example if the number of
*   audio channels or the sample rate are not compatible).
*   When returning false, it is strongly advised to print a message with
*   the "print" function for the end user using the print function.
*/
DSP_EXPORT  bool  initialize()
{
   /// write script initialization here
   /// and return false if something is not supported (number of channels, sample rate...)
   return true;
}

/** Reset the state of the filter.
*
*/
DSP_EXPORT  void  reset()
{
}

/** Returns the tail size in samples.
*   use -1 for infinite (typically for audio or MIDI generators or synths).
*   use 0 if the processor does not produce any sound when fed with silence (default).
*/
DSP_EXPORT  int  getTailSize()
{
   return  0;
}

/** Returns the latency added by the script if any, in samples.
*   Returns 0 by default.
*/
DSP_EXPORT  int  getLatency()
```


Copyright (c) 2014-2015 Blue Cat Audio

```
{
   return  0;
}

/** Per-sample processing function: called for every sample with updated parameters values.
*   If defined, the processBlock function is ignored.
*   ioSample: an array of current audio samples (one element for each audio channel).
* You can access the current sample of channel 'ch' using ioSample[ch].
*/
/*DSP_EXPORT void processSample(double ioSample[])
{

}*/

/** Update internal parameters from inputParameters array when required.
*   Called every sample, right before the processSample() method, or every block,
*   before the processBlock() method.
*   This function will not be called if input parameters have not been modified since last call.
*/
DSP_EXPORT  void  updateInputParameters()
{
}

/** Per-block processing function: called for every block with updated parameters values.
 *   Ignored if the processSample() function is defined.
 */
DSP_EXPORT  void  processBlock(struct BlockData* data)
{

}

/** Called for every block to update internal parameters from the inputParameters and
*   inputStrings arrays that have been updated by the host.
*   This function will not be called if input parameters have not been modified since last call,
*   and if transport info (tempo and time signature only) are unchanged.
*/
DSP_EXPORT  void  updateInputParametersForBlock(const struct TransportInfo* info)
{

}

/** Update output parameters values array from internal variables.
*
*/
DSP_EXPORT  void  computeOutputData()
{
}


/** Cleanup on shutdown: release all resources (allocated memory, files, system handles...).
 *
 */
DSP_EXPORT  void  shutdown()
{

}
```


Copyright (c) 2014-2015 Blue Cat Audio

Writing Your Own User Interface

Overview

In many cases you can use the default skin provided with the plug- in, as it will display all the controls and meters required for your scripts automatically. You may however build a custom user interface for your most advanced scripts: a dedicated user interface is usually much easier to use and will probably look better too.

For this purpose you can write custom skins using the KUIML language.

To build the skin, you can refer to the parameters list available later in this manual as well as the mapping.inc file provided with the default skin.

Also, to make sure that the appropriate script is loaded for your particular skin (if it is specific to a script), you can force the value of the dsp.input_string0 string to the path of your script. To make it portable, you can use the *%USER_SCRIPTS_PATH %* macro that defines the root path for the scripts in the user documents folder. If "my_script.cxx" is the name of your script, located in the root of the user scripts directory, use for example the following lines in the skin file:

```
<STRING id="forced_path" default="%USER_SCRIPTS_PATH%/my_script.cxx"/>
<STRING_LINK from="forced_path" to="dsp.input_string0" enabled="true"/>
```

This will ensure that the appropriate script file is loaded when the skin is selected. Since a skin can be set for each preset, if you have multiple scripts that require different skins, you may want to save these as presets, so that both the skin and the script are reloaded automatically.

The plug- in is delivered with multiple custom skins (available from the main menu under "Preset Skin", or loaded by some presets). They all use the default layout of the plug- in for meters and menus and add extra widgets, but you can also write an entirely dedicated user interface from scratch, without using the plug- in theme or its main layout.

Factory Skins source can also be browsed online.

Building the GUI from Scratch

Once you become familiar with Blue Cat's Plug'n Script and KUIML, you may want to build your user interface from scratch, with your own controls and layout. Given the dynamic nature of the Plug'n Script plug- in (the script and its parameters can change at runtime), here are a few tips and tricks that can be useful if you do not plan to reuse the default skins provided with the plug- in as a starting point.

Finalize the DSP first

Because DSP changes may have an impact on the exposed parameters and the user interface, and because you may not want to show the end user the dsp log file that usually reports the errors, it is recommended to finalize the audio/ MIDI processor before building the user interface. You may otherwise have a hard time maintaining the user interface while making impacting changes on the DSP part.

Use the mapping file

It is recommended to use the mapping.inc (and mapping_utils.inc) files that are delivered with the default skins:

First, these files define variables (macros) for all the data exposed by the dsp that are much easier to read than the usual 'dsp.inputXX' parameter names. This will also ensure that your custom skin can be easily modified in case the future versions of the plug- in expose more information.

Second, it also defines "custom" input and output parameters that use the characteristics defined in the DSP script (min, max, default values, units etc.), unlike the actual dsp parameters which range (0 to 100%) and name (Control 0, Control1...) are fixed.

Copyright (c) 2014-2015 Blue Cat Audio

You will want to use these custom parameters for your controls ($custom_inputX $) or meters ($custom_outputX $), so that they exactly correspond to the definitions of the dsp script:

```
<?xml version="1.0" encoding="utf-8" ?>
<SKIN author="Blue Cat Audio" name="Custom" font_face="Arial" font_size="14"
    text_color="#dddddd" background_color="#000000">
  <!-- Include mapping file -->
  <INCLUDE_ONCE file="mapping.inc"/>

  <!-- Force to load the custom script-->
  <STRING id="script_forced_path" default="%USER_SCRIPTS_PATH%/Utility/custom_script.cxx"/>
  <STRING_LINK from="script_forced_path" to="dsp.input_string0"/>

  <!-- Parameter Control Using Custom Parameter -->
  <COLUMN>
    <PARAM_TEXT param_id="custom_param0" content="{name}"  width="100%"/>
    <PARAM_TEXT param_id="custom_param0" width="65"/>
    <IMAGE_PARAM_KNOB image="my_knob.png" param_id="custom_param0"/>
  </COLUMN>
</SKIN>
```

Adding a User Manual

It is also possible (and strongly recommended) to add a user manual to the dsp script. It will be made available to the end user in the title bar, with an additional question mark icon (see the chapter about user interface for more details).

Both pdf and html manuals are supported. Simply distribute the script with an html or pdf file in the same directory as the script, with the appropriate naming convention:

- For text scripts, the manual should have the same name as the script file, with the *.html* or *.pdf* extension. For example: *myscript.cxx* and *myscript.cxx.pdf*.
- For binary scripts, the manual should have the same name as the script file (except that even on windows the file extension of the script is ignored and replaced by *.bin*), with the *.html* or *.pdf* extension. For example: *myscript.bin* and *myscript.bin.pdf* , or *my_script.x86* and *my_script.bin.pdf*.

If no manual is provided, the help icon will not be visible.

Resources

If you are new to dsp and programming, or if you just want to learn more, you will find below a few resources that you may find interesting:

Programming

- The AngelScript Language: reference for AngelScript programming.
- The Plug'n Script Repository: source code repository on GitHub to share dsp scripts with the community.

Signal Processing

- music- dsp.org: the ultimate mailing list about digital signal processing. You will also find there many links and an interesting source code archive.
- The DSP and Plug- in Development forum on kvraudio: a good forum to discuss dsp and plug- in development in general.

MIDI

- midi.org: the MIDI Manufacturers association. Several resources and tutorials about MIDI.
- MIDI 1.0: The full MIDI standard specification.

Also, be sure to check our Forum regularly and post there if you have any question.

Copyright (c) 2014-2015 Blue Cat Audio

Blue Cat Audio Plug- Ins Basics

This chapter describes the basic features that are common to all our plug- ins. If you are already familiar with our products, you can skip this part.

User Interface Basics

About Skins

Like all Blue Cat Audio plug- ins, Blue Cat's Plug'n Script uses a skinnable user interface. It means that the appearance and behavior of the user interface can be entirely customized.

Especially with third party skins, the experience may be quite different from the one offered by the default skins that we provide. However, our plug- ins and our skinning engine have several standard features that will be available whatever your favorite skin.

More information about custom skins can be found in the skins section.

The Main Toolbar

In most skins, an optional toolbar at the top of the user interface gives you access to the main options and settings of the plug- in:

Presets Area

At the center of the toolbar, you can see the current preset area (the "Default Settings" box). It displays the name of the current preset, with a "*" at the end if it has been modified since loaded.

The arrows on the left and right let you navigate thru the (factory and user) presets available for the plug- in.

Clicking on the preset name opens the presets menu which lets you manage the presets of the plug- in.

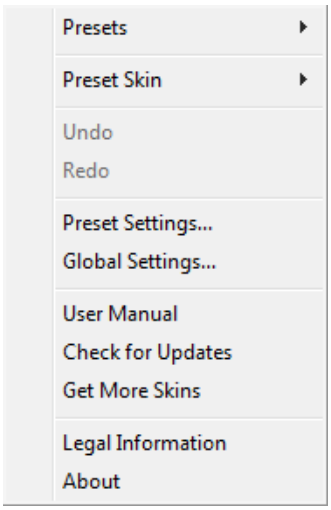
Commands

The icons in the toolbar give you access the to the following commands that are detailed in the next paragraph:

Icon	Name	Function
	Menu	Open the main menu
	Undo	Undo
	Redo	Redo
	Manual	User Manual
	About	About

The Main Menu

The main menu is available from the main toolbar, or if you right click anywhere on the background of the plug- in:



- **Presets**: opens the presets menu to manage presets.
- **Preset Skin**: opens the skins menu to choose the skin for the current preset and manage alternative skins for the software.

Copyright (c) 2014-2015 Blue Cat Audio

- **Undo/ Redo:** undo or redo the latest modifications. This includes all changes made to the current preset settings such as MIDI or automation preferences.
- **Presets Settings:** open the presets settings window. It lets you change the skin, MIDI and automation settings for the current preset.
- **Global Settings:** open the global settings window. It lets you change the skin, MIDI and automation settings that are used by default in all instances of the plug- in (if not overridden by the current preset).
- **User Manual:** open this user manual.
- **Check for Updates:** opens up our website to let you check if any update for this software is available.
- **Get More Skins:** get more skins for this software.
- **Legal Information**: browse licensing and misc legal documents.
- **About:** displays the “about” dialog box.

The Title Bar

In most skins, an icon bar is located below the main toolbar. The first commands are the same for all plug- ins:

From left to right:

- **Smooth Bypass:** smoothly bypass the plug- in (just like a power button).
- **Window Opacity:** reduce the opacity value by turning the knob, and make the plug- in window transparent. (the result may depend on the host application).
- **Show/ Hide MIDI and Automation Control Settings:** show/ hide the buttons that give access to direct MIDI/ Automation setup for each parameter in the user interface. This button will show or hide dropdown menu buttons that popup a MIDI/ Automation setup menu described here.

Controls

Examples

Here are a few examples of typical controls you will encounter in the user interface of our plug- ins:

slider	slider	knob	button	Text control

Interacting with Controls

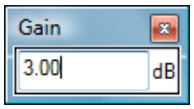
You can interact with the controls of the plug- in interface either with the mouse or the keyboard.

Setting the keyboard focus on a control (so that it responds to key strokes) may be automatic (when you pass the mouse over it it gets focus) or manual (you have to click on the control to set the focus on it). Note that all host applications behave differently regarding keyboard handling. In some applications you may not be able to use all keys described later in this manual to interact with our plug- ins. It is usually made obvious to you to know the active surfaces of the skin (the places where you can click with the mouse): the mouse cursor usually changes when you can do something on a control. In the default skins delivered with the plug- in, the cursor changes to a small hand or an arrow to tell you when your mouse is over an active control.

Mouse

Various mouse movements will let you interact with the controls:

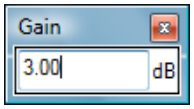
Copyright (c) 2014-2015 Blue Cat Audio

Mouse Interaction	Action
Left Click	Acquire focus and start dragging or push (button)
Left Click + Alt Key	Set the value to default
Left Double Click	Acquire focus and launch the “fine tuning” edit box (except button): 
Right Click	Set the value to default
Mouse Wheel	Increment or decrement the position (focus required)
Mouse Drag	Change the control position depending on mouse movement (except button)

Keyboard

All control widgets support the following keys (note that some of them are caught by the host and thus never forwarded to the control. For example in Steinberg Cubase SX you cannot use the arrow keys to control the plug- in):

Keys Common to All Controls

Key	Action
Up Arrow	Small increment of the position (up or right)
Down Arrow	Small increment of the position (down or left)
Left Arrow	Same as Down Arrow
Right Arrow	Same as Up Arrow
Page Up	Large increment of the position (up or right)
Page Down	Large decrement of the position (down or left)
+	Small increment of the value of the control
-	Small decrement of the value of the control
d	Set to default value (same as mouse right click)
e	Opens the 'fine tuning' window to precisely set the parameter: 
SHIFT	When the key is down, the fine tuning mode is on, and you can modify the value with better precision when moving the mouse, the mouse wheel or using the keyboard. Just release the key to get back to the normal mode.

Keys Specific to Buttons

Key	Action
Enter	Pushes the button

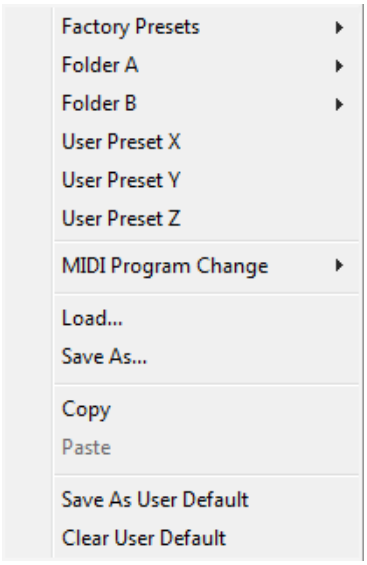
Presets

To get started with the plug- in and discover its capabilities, a couple of factory presets are provided. You can also save your own presets and recall them later for other projects. Our plug- ins propose a full- featured preset manager to let you save, browse, organize and recall its presets.

The Presets Menu

The presets menu can be opened from the main menu or the main toolbar. It displays the list of presets available for the plug- in as well as commands to load, save or organize presets:

Copyright (c) 2014-2015 Blue Cat Audio

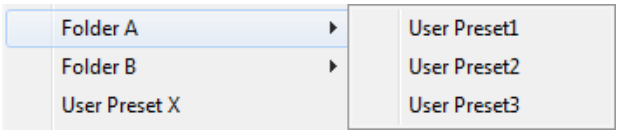


- **Factory Presets:** shows the list of factory presets delivered with the plug- in.
- **"Folder A" to "User Preset Z":** user presets and categories.
- **MIDI Program Change:** activate MIDI Program Change support (see below).
- **Load:** load preset from file.
- **Save:** save current state to last loaded user preset.
- **Save As:** save current preset to a file.
- **Copy** copy preset to the system clipboard.
- **Paste** paste preset from the system clipboard, if available.
- **Save As User Default:** save the current state as the default preset. This preset is used every time a new instance of the plug- in is created.
- **Clear User Default:** reset the default preset to its factory state: this makes the plug- in forgets the custom settings you might have saved as a default preset.

More about Presets

There are two types of presets: factory presets (read only) that are provided with the plug- in, and user presets that can be created and stored by the user.

The user presets are stored in a subdirectory of the documents folders of your profile ("Documents" on Mac, and "My Documents" on Windows): Blue Cat Audio/ [Plug- in Name]/ Presets. Each preset is stored as an individual file. You can create folders and subfolders in the Presets directory to classify your presets, as shown in the example below:

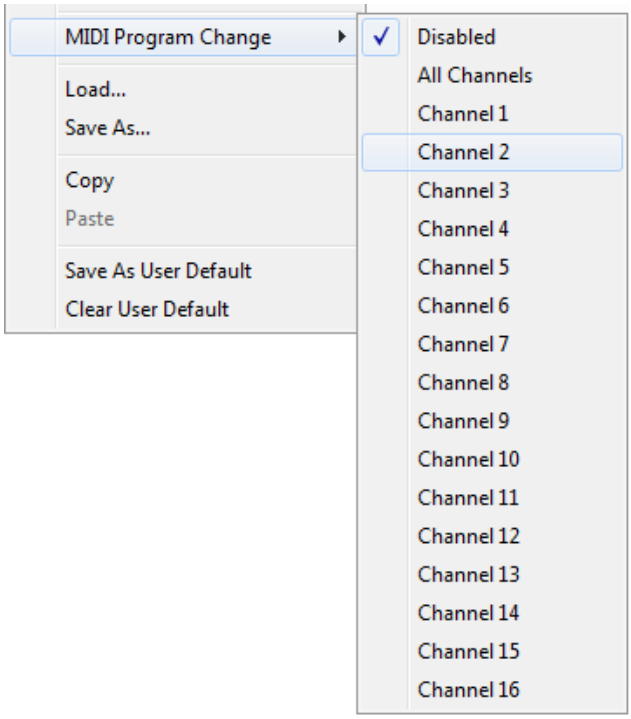


If you save a preset named "Default" in the root Presets directory, it will override the factory default preset (that's what "Save As Default" does). To restore the factory default preset, you can just remove this file or use the "Reset Default" command.

MIDI Program Change

It is possible to load presets remotely using MIDI "Bank Select" and "Program Change" messages. To enable this feature, select a MIDI channel to receive the events from in the MIDI Program Change menu item from the presets menu:

Copyright (c) 2014-2015 Blue Cat Audio



This setting is a global preference (shared accross all instances of the plug- in). Once activated, the plug- in menu will display the bank number followed by the preset number for each preset:

Every root folder is considered as a new bank, starting with the factory presets (bank 0). Program and bank numbers may change while you add folders and presets, so you should be careful when naming them if bank and program numbers matter to you. It is recommended to use folders to make this task simpler. As a side note, sub folders do not define additional banks (all presets contained in sub folders are associated with the current bank.

As specified by MIDI, bank select messages are not used until a program is actually selected.

MIDI Implementation note: the software supports all types of Bank Select methods. You can use either MIDI CC 0 or MIDI CC 32 to select banks. If both are used simultaneously, they are combined together so that you can use more banks (in this case CC0 is LSB and CC32 is MSB, and actual bank number is 128*CC0+CC32).

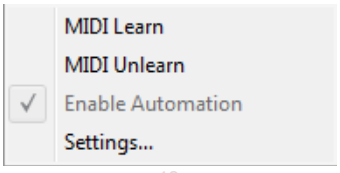
MIDI and Automation Control

Blue Cat's Plug'n Script can also be remotely controlled via MIDI using MIDI CC ("Control Change") messages or automation curves, if your host application supports it. It is possible to customize the channel, control numbers, range and response curve used for each parameter in the settings panel available from the main menu (see the Plug- in Settings chapter for more details).

MIDI and Automation Settings Menu

Most skins also provide the ability to change MIDI and automation settings for each parameter directly in the main user interface. When this feature is activated using the corresponding title bar button, dropdown menu buttons appears next to main controls:

Clicking on this button shows the MIDI/ Automation settings menu:

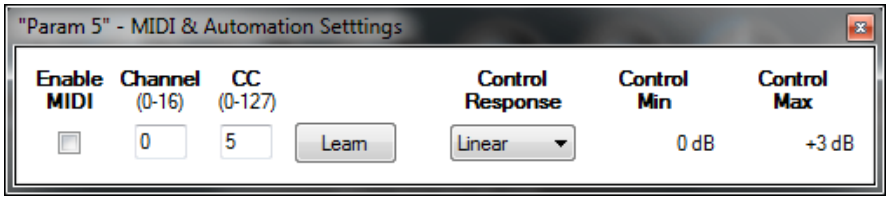


Copyright (c) 2014-2015 Blue Cat Audio

- **MIDI Learn:** launches MIDI learn mode for the control: touch your MIDI controller and the control will learn from it the MIDI channel and CC number. To end the learn mode, reopen this menu and deselect the option.
- **MIDI Unlearn:** deactivates MIDI control for this parameter.
- **Settings:** launches the advanced settings panel described below. This controls the settings for the current preset.

Advanced MIDI and Automation Settings

You can completely customize the way the plug- in is controlled by automation and MIDI. For a global view of all parameters at a time, you can use the Plug- in Settings window for the current preset which is available from the main menu.



MIDI Settings:

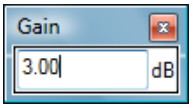
- **Enable MIDI**: enable/ disable the MIDI control of the parameter.
- **Channel**: MIDI Channel for the parameter control. If set to 0, the plug- ins will accept Control Change Messages from all MIDI Channels (MIDI Omni mode).
- **CC**: Control Change Number.
- **Learn**: click on this button to activate the MIDI learn functionality. When it is activated, you can move your MIDI controller, and the plug- in will automatically set the MIDI Channel and CC Number.

MIDI and automation settings:

- **Response**: response curve of the MIDI or automation control: from very fast to slow control.
- **Min**: minimal value of the parameter when MIDI controlled or automated.
- **Max**: Maximum value of the parameter when MIDI controlled or automated.

Note: if the Min value is higher than the Max value, the response curve will be reversed: increasing the control value will decrease the parameter value.

Note: if you double click on the parameter text control boxes for the max and min values, a “fine tuning” edit box will appear and let you change the min and max values with more precision:

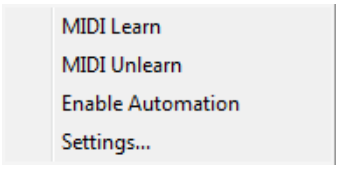


MIDI and Automation Output

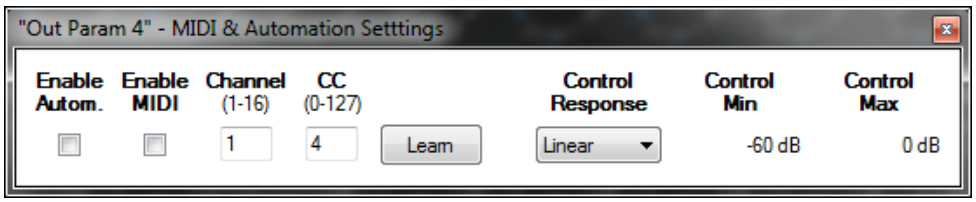
Blue Cat's Plug'n Script can also generate MIDI CC Events or automation curves thanks to its output parameters.

You have access to the same settings for the output parameters as you have for input parameters using the dropdown menu, except that you can also enable or disable automation.

The output parameters MIDI and Automation Menu:



The advanced output parameters MIDI and Automation settings window:



More

Copyright (c) 2014-2015 Blue Cat Audio

Check our online tutorial for more screenshots and more examples of our plug- ins user interfaces.

Copyright (c) 2014-2015 Blue Cat Audio

Blue Cat's Plug'n Script Parameters

All parameters described below can be automated and controlled via MIDI if your host application supports it. You can precisely define this behavior in the settings panels described later in this manual.

Input

The input parameters of this plug- in are the following:

Name	Unit	Description	Comment
General			
Bypass		Bypass the effect.	
Reload Script		Value changes trigger the reload the current script.	
Enable Meters		Enable or disable audio level meters.	
5 Reserved Parameters		5 reserved parameters for future usage. Do not use.	
For each script input parameter (index: i) - 48 parameters			
Param i	%	Input parameter number i for the script (actual range is [0-100%], whatever the internal range selected in the script).	

Output

The output parameters of this plug- in are the following:

Name	Unit	Description	Comment
For each script output parameter (index: i) - 32 parameters			
Param i	%	Output parameter number i for the script (actual range is [0-100%], whatever the internal range selected in the script).	

Copyright (c) 2014-2015 Blue Cat Audio

Plug- in Settings

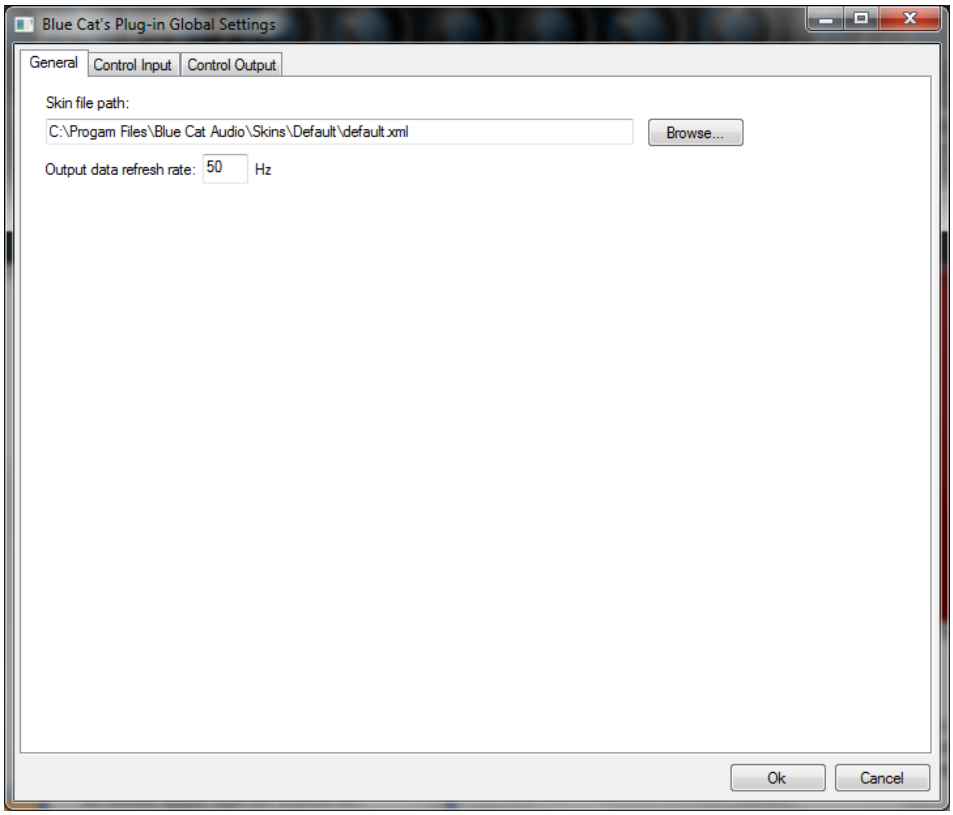
In addition to the controls offered in the main user interface, Blue Cat's Plug'n Script has various settings that let you fine tune the behavior of the plug- in. You can choose to change these settings either for the current preset or globally for all instances of the plug- in.

The Global Settings Window

The settings available in this window ***apply to all instances of the plug- in, for all presets***, if not overridden in the presets settings. Consider these settings as “default” settings.

General

You can change the default skin for all instances of the plug- in: write the skin file path in the text edit box or click on the button to open a file chooser dialog. If you have several instances of the plug- in opened in your session, you will have to re- open the user interfaces of these plug- ins to see the skin change.



The output data refresh rate can also be customized for all instances of the plug- in. It controls the refresh rate of non- audio data produced by the plug- in (parameters, curves...). It also controls the refresh rate of output MIDI CC messages or output automation data. The higher the refresh rate, the better precision, but also the higher cpu usage (some host applications may also have trouble recording MIDI data at high refresh rates). The default value is 50 Hz.

Global Control Input Settings (MIDI and Automation)

The plug- in offers a couple of settings that affect the way it is controlled by MIDI messages or automation. While the first settings only apply to MIDI control, the "Control Response", "Min" and "Max" settings apply to ***both automation and MIDI control***.

For each parameter you can define a default MIDI channel and CC number. You can then control the plug- in with an external MIDI controller or one of our plug- ins that generate MIDI messages.

The settings below are available for each plug- in parameter.

MIDI Settings:

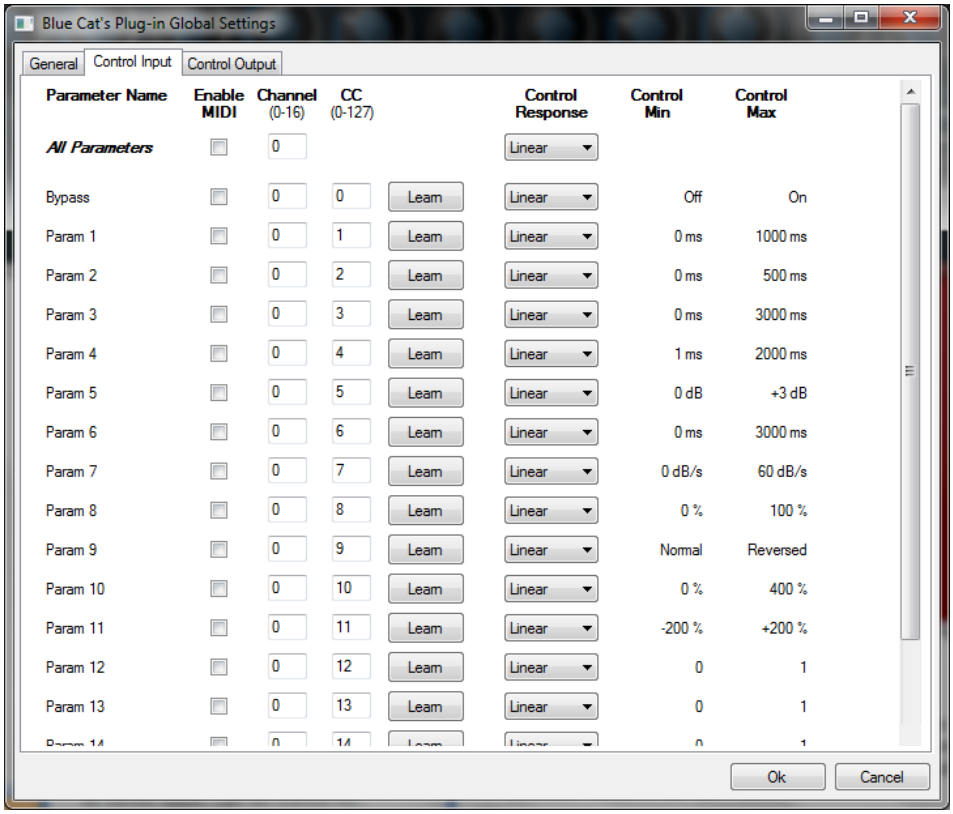
- **Enable MIDI**: enable/ disable the MIDI control of the parameter.
- **Channel**: MIDI Channel for the parameter control. If set to 0, the plug- ins will accept Control Change Messages from all MIDI Channels (MIDI Omni mode).
- **CC**: Control Change Number.

Copyright (c) 2014-2015 Blue Cat Audio

- **Learn**: click on this button to activate the MIDI learn functionality. When it is activated, you can move your MIDI controller, and the plug- in will automatically set the MIDI Channel and CC Number.

MIDI and automation settings:

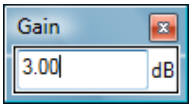
- **Response**: response curve of the MIDI or automation control: from very fast to slow control.
- **Min**: minimal value of the parameter when MIDI controlled or automated.
- **Max**: Maximum value of the parameter when MIDI controlled or automated.



(generic screen shot, does not correspond to the actual plug- in parameters)

Note: if the Min value is higher than the Max value, the response curve will be reversed: increasing the control value will decrease the parameter value.

Note: if you double click on the parameter text control boxes for the max and min values, a “fine tuning” edit box will appear and let you change the min and max values with more precision:

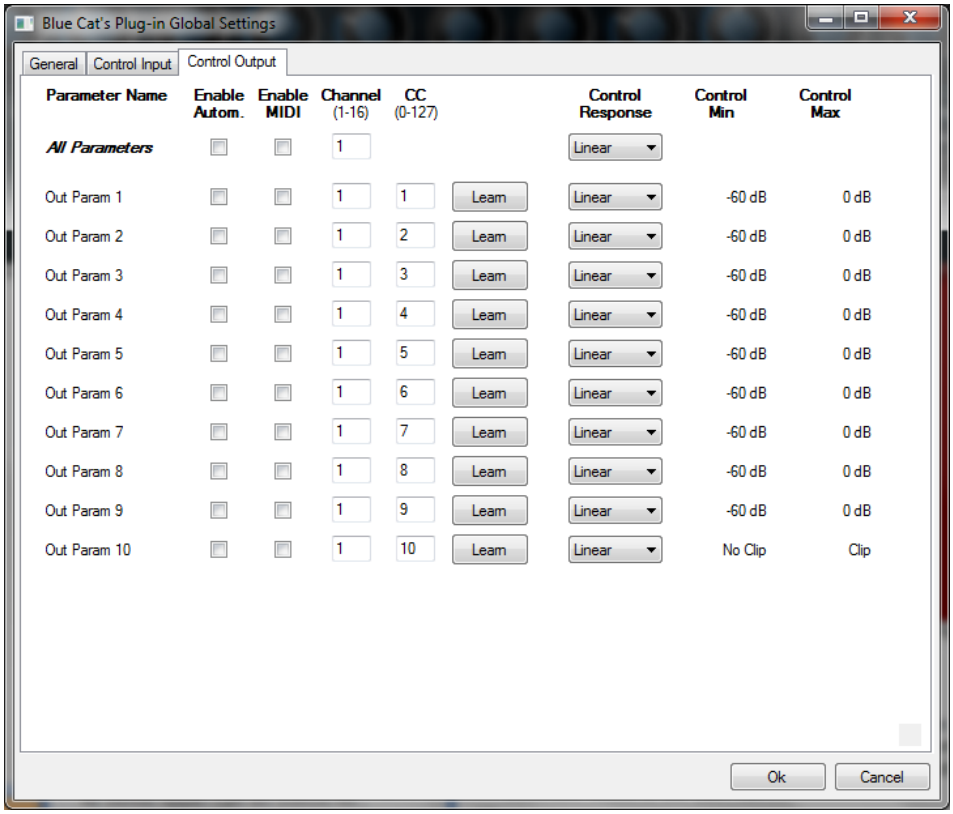


Global Control Output Settings (MIDI and Automation)

You can set the same properties for the output parameters as for the input parameters: in this case, they may trigger MIDI CC messages or generate automation curves when modified. Since it's output, you cannot set the channel to MIDI Omni, so you must choose a channel.

Output parameters can also generate automation curves in most host applications. You can enable automation for any output parameters you are interested in (see the "Enable Autom." checkbox).

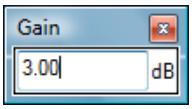
Copyright (c) 2014-2015 Blue Cat Audio



(generic screen shot, does not correspond to the actual plug- in parameters)

Note: if the Min value is higher than the Max value, the response curve will be reversed: increasing the control value will decrease the parameter value.

Note: if you double click on the parameter text control boxes for the max and min values, a “fine tuning” edit box will appear and let you change the min and max values with more precision:



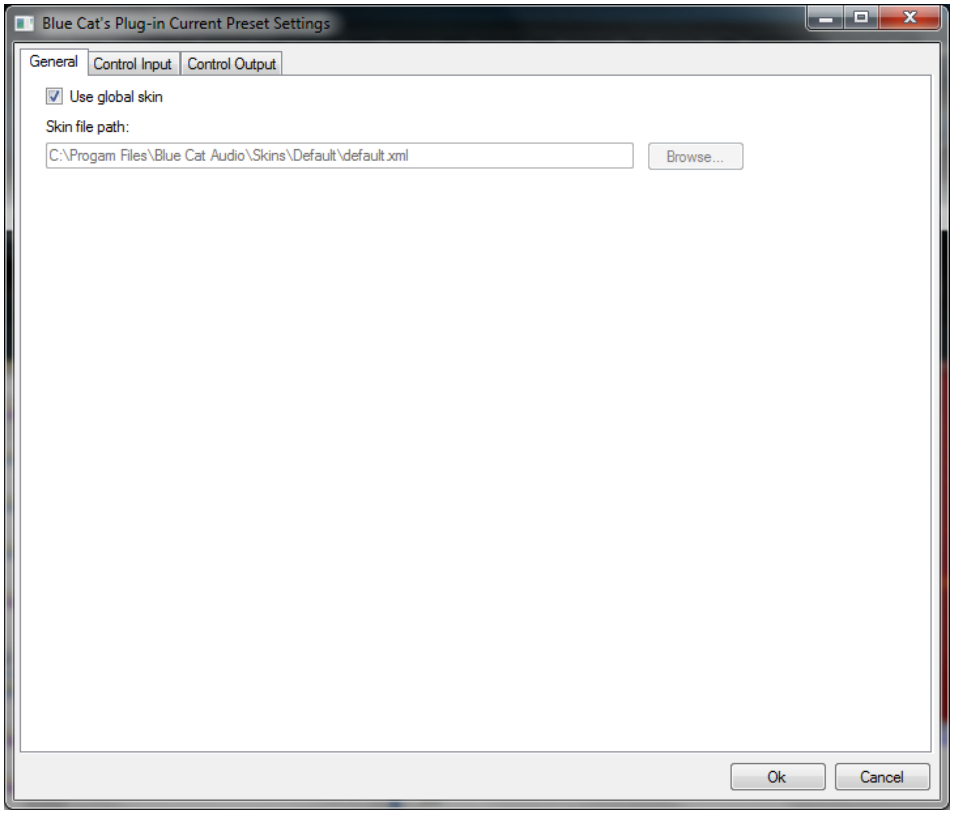
The Current Preset Settings Window

In this window you can change the settings ***for the current preset of the current instance of the plug- in only***.

Preset Skin

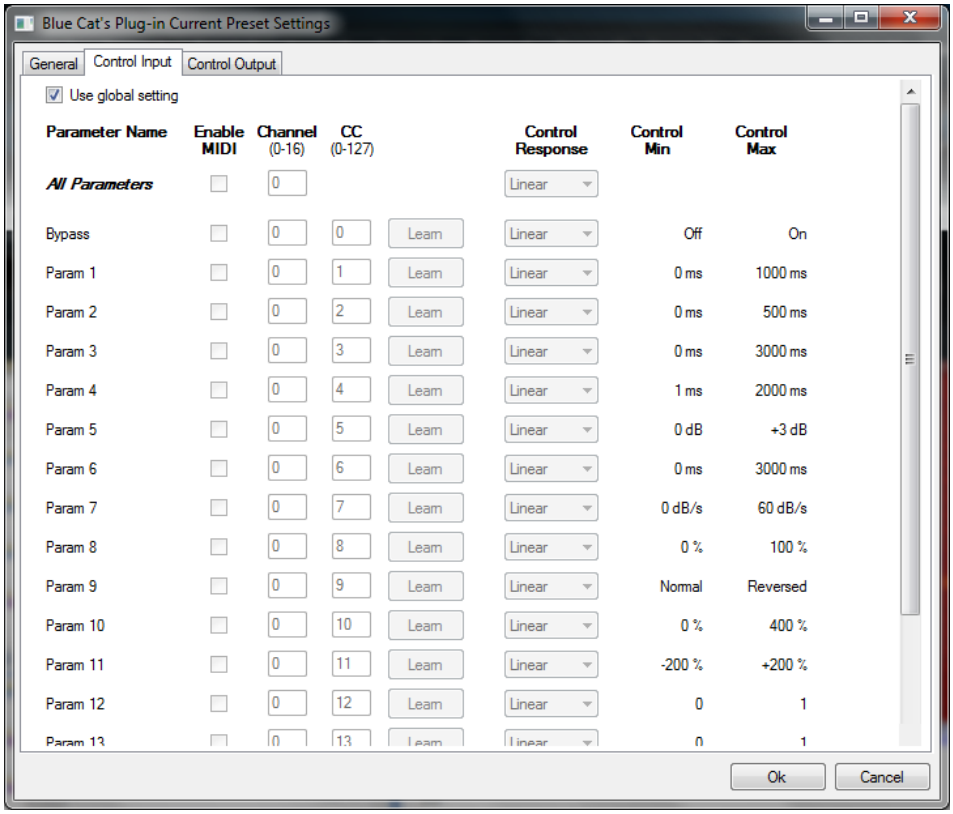
You can choose to use the global skin setting or to change the skin for the current preset. This way you can have different skins for different instances of the plug- in in the same session in order to differentiate them.

Copyright (c) 2014-2015 Blue Cat Audio



Preset Control Input Settings (MIDI and Automation)

Use the global settings or override them for the current preset. The parameters are the same as for the global input settings.

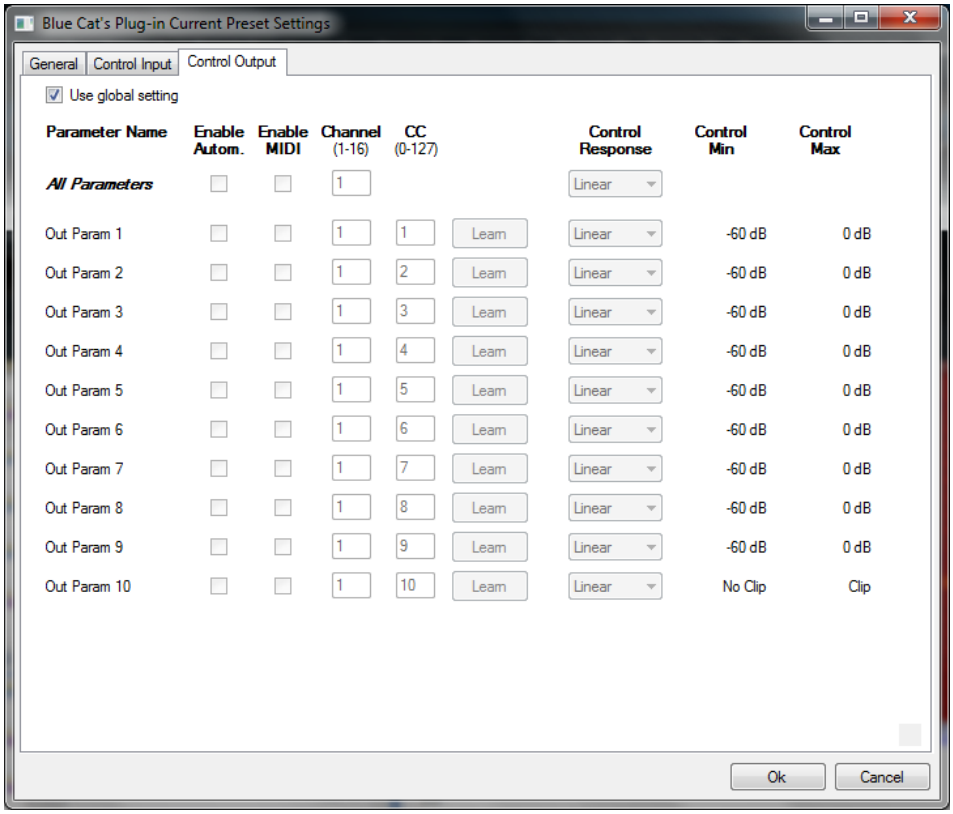


(generic screen shot, does not correspond to the actual plug- in parameters)

Preset Control Output Settings (MIDI and Automation)

Use the global settings or override them for the current preset. The parameters are the same as for the global output settings.

Copyright (c) 2014-2015 Blue Cat Audio



(generic screen shot, does not correspond to the actual plug- in parameters)

Copyright (c) 2014-2015 Blue Cat Audio

About Skins

Blue Cat's Plug'n Script integrates Blue Cat's skinning engine that allows you to customize the user interface. You can download alternate skins for your plug- in at the following address:

http:// www.bluecataudio.com/ Skins/ Product_ PlugNScript

If you don't find a skin that fits your need or if you want a custom one, you can choose to create your own skin.

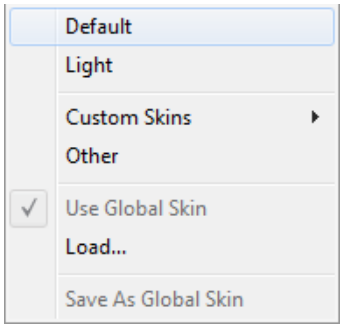
Choosing the Skin

There are two ways to select the skin of your plug- in: you can change the default (or 'global') skin, or change the skin for the current preset only (either in the preset settings page or from the main menu). The global skin applies to all plug- in instances (choose this one if you want to use the skin used by default, regardless of the session or preset), whereas the current preset skin only applies to the current preset of the current plug- in instance (use this one if you want to change only the skin for the current session/ preset).

Note: in some host applications, the plug- in window won't resize automatically when you choose a skin with a different size. In this case, just close the window and re- open it: it will be displayed with the right size.

The Skins Menu

The skins menu can be opened from the main menu. It displays the list of skins available for the plug- in as well as commands to manage the skin used by default when no preset skin has been selected:



- **First Section - Factory Skins:** shows the list of factory skins delivered with the plug- in ("Default" and "Light" in this example).
- **Second Section - User Skins:** shows the list of user skins that have been installed in the Documents Skins folder for the plug- in (see below).
- **Use Global Skin:** use the global skin for the current preset/ session (unloads any custom skin previously selected for the current preset).
- **Load:** opens a file browser dialog to manually select the skin from the file system.
- **Save As Global Skin:** use the current preset skin as the global skin (loaded by default if no preset skin has been defined).

Installing User Skins

If you want to be able to select user skins directly from the skins menu, you simply need to install them in the Skins directory available in the plug- in's documents folder:

[Your Documents Folder]/ Blue Cat Audio/ [Plugin Name]/ Skins/

The skin engine will scan this folder for new skins (xml files) and display them in the menu. *The skin files should be in the root skins folder or in a subdirectory inside this folder: subdirectories are not scanned recursively*.

Other Methods to Select Skins

You can also select the skins in the settings panels available from the main menu:

The global skin (used by default if no preset skin has been selected) can be changed in the global settings pane. The current preset skin can be changed in the preset settings page.

Create a Custom Skin

Copyright (c) 2014-2015 Blue Cat Audio

You can create custom skins for your plug- in in order to adapt it to your exact needs. You can change its look and feel and make it completely integrated in your virtual studio!

Just read the Blue Cat's Skinning Language manual and download the samples for the tutorial on http:// www.bluecataudio.com/ Skins. You can get ready to create your own skins in a few minutes.

You can then share your skins on our website.

Copyright (c) 2014-2015 Blue Cat Audio

FAQ

Automation Output

When I use the plug- in, my host application is slowed down and I get audio drop- outs. How can I solve this problem?

If your host application does not let you select the parameters for which you want to record automation, it might be doing some extra work for handling the output parameters generated by the plug- in. A solution is to disable the output parameters you do not use in the Preset or Global settings/ Automation Output pane.

When I use the plug- in, my host application draws many automation curves on the current track and yet I have not touched any control on the plug- in. How can I avoid that?

If your host application does not let you select the parameters for which you want to record automation, it will record automation for all the output parameters generated by the plug- in. You can disable the output parameters you do not use in the Preset or Global settings/ Automation Output pane.

I am trying to record automation curves as shown in the tutorials but it does not work. What do I have to do?

If your host application lets you select the parameters for which you want to record automation, check that the parameters you want to record are enabled. You also need to check that the plug- in automation output is enabled for these parameters. Check the Preset or Global settings/ Automation Output pane and select the parameters you want to record.

When I use the plug- in in a project, every time I save the project, my application shows it as unsaved right away. How can I solve this issue?

This plug- in produces automation output, and some host applications consider that any change to one of the output parameters is a change for the project. In order to solve this problem, and if you do not use the automation output capability of this plug- in, you can disable automation output from the plug- in settings panel (Global or Preset Settings/ Automation Output pane: uncheck all parameters).

MIDI Output

How can I enable the MIDI output of the plug- in?

If your host application supports it, you can choose to send MIDI CC messages based on the output parameters computed by the plug- in. The feature is not enabled by default, so you need to go to the current preset settings (of the global settings if you want to enable this for all instances - not recommended) and open the "MIDI output" pane. For each parameter you are interested in, enable MIDI output and select the appropriate channel and CC number. The plug- in will then send MIDI messages to the host application. Check our tutorials for a detailed explanation of MIDI output routing in your favorite host.

Plug- ins Formats

What are DirectX (DX), Audio Unit (AU), RTAS, AAX and VST plug- ins?

VST, Audio Unit, RTAS, AAX and DirectX plug- ins are software components than can be used in "Host" Software (such as Blue Cat's PatchWork, Pro Tools, Cakewalk Sonar, Steinberg Cubase or Wavelab, Sony Vegas, Logic Pro, Garage Band, Ableton Live...) in order to perform some MIDI and/ Or Audio real- time Processing tasks. To be more precise what we usually call directX plug- ins is actually a "DirectShow Filter". VST is owned by Steinberg, DirectX by Microsoft while Audio Units is the property of Apple Computers and RTAS and AAX are owned by AVID.

How can I use your plug- ins?

Download and install a host software, then download and install the plug- ins from our page. They will appear in the host "effects" menu. If you are using a DirectX host with our MIDI controllable plug- ins and they do not show up in you host list, you might need to use our freeware DXi Manager. Note that our plug- ins are sometimes shown in the "MIDI controllable audio effects" or "soft synths" sections in some host applications.

Where can I find a host?

You use Blue Cat's PatchWork standalone application to host any VST or Audio Unit plug- in for real- time operation. There are also many commercial or freeware hosts that will suit your needs. You can find a long list of applications here. You can also use demos of Cubase, Wavelab, Ableton Live, Tracktion, or Sonar software, which are usually available on the companies websites. On the Mac, Garage Band is part of the system and can be also used to host our Audio Unit plug- ins.

What are the main differences between DX, VST, RTAS, AAX and Audio Units?

Copyright (c) 2014-2015 Blue Cat Audio

VST is a protocol that works on several platforms (Mainly Windows, MacOS, BeOS, and some Linux platforms use it as well) whereas DirectX is Microsoft Windows only, and Audio Units are available only on the Mac. RTAS and AAX are available on Mac and PC, but only for Pro Tools products (AVID). AAX plug- ins are compatible with Pro Tools 10 and higher, whereas RTAS versions are compatible for Pro Tools 10 and lower versions.

What is the difference between DXi and DX effects?

DXi effects are MIDI controllable DirectX plug- ins. It's the same as DX effects plus MIDI control.

Which version (VST, Audio Unit, RTAS or DX) should I use in my host application?

Here is a list of host software and the version we recommend you to use. Many other applications are supported, check your application user manual to choose the best version (AU stands for "Audio Unit" and DX for "DirectX"):

Host Application	Plug- in Version
Cakewalk Products (Sonar, Project5...)	VST for Sonar 8 and newer, DX for older
Sony Products (Vegas, ACID, Soundforge)	VST
Steinberg Products (Cubase, Nuendo, Wavelab)	VST (Mac or PC)
Ableton Live	VST (Mac or PC)
Adobe Audition	any
EnergyXT	VST
Magix Samplitude	VST
Avid (Digidesign) Pro Tools	RTAS for PT 9 and earlier, AAX for PT 10 and newer
Apple applications (Logic Pro, Logic Express, Garage Band...)	AU
Cockos Reaper	VST (Mac or PC)
...	

In general, we recommend using VST over Audio Unit on Mac (when both are supported), particularly for the plug- ins that can output automation or MIDI CC messages.

Where can I find more information about this topic?

- www.steinberg.net
- www.microsoft.com
- www.kvraudio.com
- www.apple.com
- www.digidesign.com

Installation

I have installed my software and it is not listed as an application. Why? What can I do? How can I launch it?

Most software you can purchase on this website is plug- ins for host applications. It means you need another application to use it. See the "Plug- ins Formats" section in this FAQ for more information.

I have installed my plug- in and it does not show up in my host application. What can I do?

First check that you application supports DirectX, Audio Unit or VST plug- ins.

If you are using the DirectX version, check that your host application supports DXi plug- ins (MIDI enabled DirectX plug- ins). If it does not, it may remove it from the DirectX plug- ins list (some applications such as Sony Vegas 5 and Cool Edit Pro are known to do this). In this case, use our DXi Manager free utility and disable the MIDI capabilities of the plug- in. You may need to reinstall the software again before it shows up into your host application. For more information about this topic please read our DXi in Sony Vegas tutorial (it is applicable to other applications than Sony Vegas).

Copyright (c) 2014-2015 Blue Cat Audio

If you are using the Direct X version and your host application supports DXi, check that the plug- in does not appear in the 'virtual synth' or 'synthesizers' category. If you wish not to use the MIDI capabilities of the plug- in and use it as a regular Direct X plug- in, you can download the freeware DXi Manager and disable the MIDI capabilities of the plug- in.

I have installed my plug- in on windows 64- bit, checked the items above, and it still does not show up in my host application. What can I do?

You are probably running a 32- bit application. If that's the case, you need to install the 32- bit version of the plug- in: the choice for 32 or 64- bit is not determined by the operating system (64- bit windows can run both 32 and 64- bit applications), but by the host application that you are using.

Why are there two plug- ins called xxxx(Mono) and xxxx(Stereo) installed ?

"Mono effects" (which are effects that do not make any difference between left and right channels) are bundled this way for performance reasons. Some developers just deliver a stereo Version which also processes (twice) mono streams. The "(Mono)" plug- in is to be used with mono streams, and the (Stereo) one with stereo streams. The effect itself is the same in both cases but the number of processed channels is different: this may drastically improve performance for CPU intensive plug- ins.

How do I uninstall my plug- ins?

Open the Windows configuration panel/ Add Remove Programs, and choose 'Remove' on the corresponding plug- in. If you have installed the VST plug- in zip file version, just delete the corresponding dlls.

Why do your plug- ins need an installer on Windows? Do they modify the registry or system settings? Why not providing just a dll?

Our plug- ins require several additional files for default settings, skins and miscellaneous data. We provide an installer for our customers' convenience. Our installers do not modify the system settings or the Windows registry, except for the DirectX versions registration. Our installers won't harm your system.

Software Version

How do I know the version of the plug- in I am using?

You can see the version information in the about box: right click on the background of the plug- in user interface and select "About". The product version is also available in the Windows Add/ Remove Programs Window.

Why do newer versions of a plug- in override older ones (they have exact same name and IDs)?

Because newer versions improve previous ones. When a new version is released, older versions are not supported anymore. In some cases the new versions may be installed as new products (compatibility reasons, major changes...), but it is explicitly documented on the product page.

A new minor version of a product I purchased some time ago is available. Where can I download the update?

The new version can be downloaded from the same place where you downloaded the original version. All information is contained in the email you received when you purchased the product. Your registration number has not changed either.

Upgrade

A new minor version of my plug- in has been released. Where can I download it?

When you purchased the plug- in, an email containing the information to download and register your software has been sent to you. You can download the new version from the exact same location as the first time.

I have downloaded a new version, do I need to uninstall the previous version?

No, you don't, except if it's specified on the product page, in the "history" section. Just run the installer and it will upgrade your software.

Automation

What is "Parameters Automation"?

The parameters of an effect can be automated in most host software. It means that you can record the changes you do during playback or recording so that it's replayed when playing back again. It's a way of sequencing parameters changes the same way you do with Audio or MIDI data.

What is "Smooth Update"?

Copyright (c) 2014-2015 Blue Cat Audio

When changing parameter values in real time or replaying a song where parameters have been automated, our plug- ins compute intermediate values between parameter changes in order to avoid "clicks" and "pops" that may occur otherwise. It results in a non audible smooth parameters update and lets you freely use automation or MIDI control to change the effects in a song.

Do your plug- ins support MIDI Control?

Yes they do. They offer precise control over the mapping of MIDI Control Change (CC) messages to parameter values: you can adjust both the channel and CC number as well as the response curve of the control. Since some of our plug- ins are also capable of creating MIDI CC from the audio signal, they can all be connected for real time signal- dependent audio effects. See our MIDI Control in Sonar tutorial for an example.

Copyright (c) 2014-2015 Blue Cat Audio

More...

This manual only covers the basics of Blue Cat's Plug'n Script. Our website offers many additional resources for your Blue Cat's Plug'n Script plug- in and is constantly updated, so keep an eye on it! You will find below a few examples of available resources.

Extra Skins

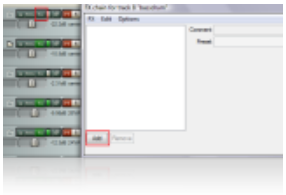
We encourage our customers to propose their own skins for our products and we often propose alternative skins to let you choose the one that best suits your needs. You can check Blue Cat's Plug'n Script skins page to get the latest skins.

There were no extra skins for Blue Cat's Plug'n Script when initially released. Please check the skins page to see if new skins are now available.

Tutorials

Many Tutorials are available on our website. They cover a wide range of topics and host applications. You will find below a list of tutorials that are related to the Blue Cat's Plug'n Script plug- in. An updated list is also available online.

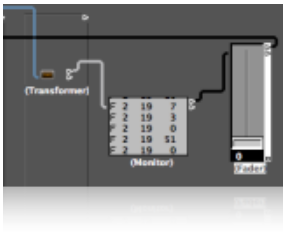
Tutorial - Using Blue Cat Audio plugins connectivity in Reaper



This tutorial shows how to use our plugins MIDI connectivity within Reaper. We use Blue Cat's Dynamics plugin as an example, but you can use any other audio plugin which provides output parameters, su...

[Read More...]

Tutorial - Connecting plug- ins together in Logic for advanced side chain effects



This tutorial shows how to use the connectivity capabilities of our plugins within Logic. We use Blue Cat's DP Meter Pro plugin (DPMP) as an example, but you can use any other audio plugin which provi...

[Read More...]

Tutorial - Using Blue Cat Audio plug- ins connectivity in Pro Tools

This tutorial shows how to use the MIDI connectivity of our plugins within Pro Tools. We use Blue Cat's DP Meter Pro plugin as an example, but you can use any other audio plugin which provides output ...

[Read More...]

Tutorial - How to Use the Free Presets Available on this Website

This tutorial shows you how to get and load free presets for your favorite Blue Cat Audio plugin. This tutorial uses Blue Cat's Dynamics for the demonstration but is applicable to all our plugins...

[Read More...]

Tutorial - Using MIDI Connectivity in a Modular Host (Plogue Bidule)

This tutorial shows you how to use the MIDI connectivity of our plugins in a modular host such as Plogue Bidule. We are using the Mac version of Bidule in this tutorial. The scenario here is very sim...

[Read More...]

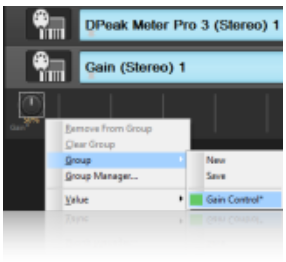
Tutorial - Using MIDI Control in Sonar

Copyright (c) 2014-2015 Blue Cat Audio

This tutorial will help you understand how to use parameters MIDI control in Cakewalk Sonar. It is applicable to any DXi plug- in (MIDI enabled DirectX plug- in)....

[Read More...]

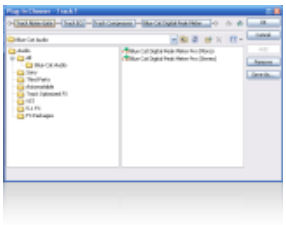
Tutorial - Connecting Plug- ins with Groups in Cakewalk Sonar



This tutorial shows how to reuse the envelopes generated by our plugins with output parameters in Cakewalk Sonar using groups. Other techniques exist (using MIDI or Automation curves), but this one ha...

[Read More...]

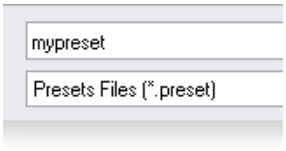
Tutorial - Using DXi plug- ins in Sony Vegas



This tutorial will teach you how to use any DXi plug- in in Sony Vegas: Sony Vegas does not support MIDI automation, so if your plug- in supports the MIDI Dxi protocol, it is removed from the plug- ins l...

[Read More...]

Tutorial - Export Plugin Presets to Share them with the Community



This tutorial shows you how to export your presets in order to share them with the community on the Blue Cat Audio website. This tutorial uses Blue Cat's Stereo Triple EQ for the demonstration bu...

[Read More...]

Tutorial - Getting Started with Plug- ins User Interface

Get used to the user interface features of Blue Cat Audio products! They have been designed for an optimal workflow, so here are the tips and tricks to save time. All these features are explained in d...

[Read More...]

Tutorial - Automation Envelopes Generation in MOTU's Digital Performer 7

This tutorial shows how to reuse the envelopes generated by our plugins with output parameters in MOTU's Digital Performer 7 host software. As of today Digital Performer does not receive MIDI events f...

[Read More...]

Updates

As you can see in the history log below, we care about constantly updating our products in order to give you the latest technology available. Please visit our website often to check if Blue Cat's Plug'n Script has been updated, or subscribe to our Newsletter to be informed of the latest news about our products.

You can also follow us on twitter and facebook for almost real time updates notification, and subscribe to our YouTube channel to see the latest videos about our software.

Copyright (c) 2014-2015 Blue Cat Audio

Versions History

V2.0 (2015/11/25)

- External side chain and auxiliary outputs support (up to 16 audio outputs).
- Native C/ C + + scripts support for optimal performance and flexibility.
- KUIML 2.0 user interface language support to build custom user interfaces, with additional scripting capabilities.
- New DSP scripts, delivered with full angelscript and C + + source code.
- 21 new custom skins included.
- User documentation (html or pdf) can now be added to scripts.
- Custom skins can be selected directly from the main menu.
- Presets can now be selected using MIDI Program change and bank select messages (can be activated with the presets menu/ MIDI Program Change item).
- Last loaded preset can now be saved directly from the presets menu without having to select the file.
- VST: fixed latency compensation reporting in Cubase (no need to re- enable the plug- in to update the latency anymore).

V1.1 (2014/10/01)

- Audio Unit can now be loaded as a MIDI effect in Logic Pro X.
- The user interface now displays switch buttons for boolean parameters instead of knobs.
- Discrete parameter values can now be selected from a dropdown menu.
- New API: scripts can now use plain strings as user input.
- New API: scripts can now read and write files.
- New scripts and presets: added audio player and recorder scripts and a wave file utility class.
- Developers can now share their scripts on a dedicated GitHub repository.
- Fixed: implemented pass- thru for MIDI events when no processing function is declared (default script for example).
- Fixed: Audio Unit Synth version did not appear as an Instrument.
- Fixed: script was reloaded when starting playback in Reaper, and the effect was not applied immediately.
- Fixed: RTAS version could freeze Pro Tools 9 and earlier at startup.

V1.0 (2014/09/18)

First release.

Thanks again for choosing our solutions!

See you soon on www.bluecataudio.com!

Copyright (c) 2014-2015 Blue Cat Audio